1

AD-A215 880

DETERMINATION OF THE UNDERLYING
TASK SCHEDULING ALGORITHM
FOR AN ADA RUNTIME SYSTEM

THESIS

Gary Alen Whitted
Captain, USAF

AFIT/GCS/ENG/89D-18

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GCS/ENG/89D-18

DETERMINATION OF THE UNDERLYING
TASK SCHEDULING ALGORITHM
FOR AN ADA RUNTIME SYSTEM

THESIS

Gary Alen Whitted
Captain, USAF

AFIT/GCS/ENG/89D-18

AFIT/GCS/ENG/89D-18

Abstract

The purpose of this thesis investigation was to determine whether the task scheduling algorithm of an Ada compiler could be detected using a suite of Ada programs. This was done by identifying the task parameters and algorithm characteristics which differentiate one scheduling algorithm from the others. After these parameters and characteristics were identified, a set of test cases was developed to encompass the various parameter relationships required to detect the execution of individual algorithms. These test cases were modeled using Ada programs. Then, the programs were compiled and executed using several Ada compilers where the task scheduling algorithms of five run-time systems was known. The execution results were analyzed to determine whether the Ada programs were capable of revealing the task scheduling algorithm used by the Ada run-time system. This analysis showed that the detection of five scheduling schemes is possible using a single Ada program. Recommendations are made to improve the current Ada program leading to an automated tool in which the user analysis could be removed.

AFIT/GCS/ENG/89D-18

DETERMINATION OF THE UNDERLYING TASK SCHEDULING ALGORITHM FOR AN ADA RUNTIME SYSTEM

THESIS

Presented to the Faculty of the School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science in Computer Engineering

Gary Alen Whitted, M.B.A, B.S.E

Captain, USAF

December, 1989

Acknowledgments

I wish to thank a number of people who helped me during the research on this thesis. In particular, several members of the Language Control Facility (LCF) at the Aeronautical Systems Division (ASD) Communications-Computer Systems Center provided invaluable assistance with the Ada program compilation and execution phase of this thesis. Mr Tom Stripe provided me with several hours of invaluable help on the LCF VAX computer. Mr Bobby Evans and Steve Wilson provided encouragement, support, and use of their computers throughout the final months of this effort.

My AFIT classmates also provided encouragement and assistance. In particular, Capt Steve March helped me with some difficult and elusive problems with developing Ada programs for my test cases.

I also wish to thank my thesis committee. Dr. Gary Lamont provided me with some valuable insight into scheduling algorithm theory. Major David Umphress provided me with a wealth of knowledge about the Ada language, in particular, Ada tasking constructs. A special thanks to my thesis advisor, Major James Howatt. He provided encouragement and editing support throughout this document. And his guidance and critical analysis helped me to achieve a successful thesis conclusion.

I want to thank my children for their prayers and support even though I spent many hours away from home working on this thesis. A special thanks to my precious wife, Carol, for her patience, prayers, support, and understanding throughout this whole ordeal.

Finally, I want to thank my Lord and Saviour, Jesus Christ for the wisdom, knowledge, understanding, and strength which He continually provides to me. Completing this thesis has helped me fully appreciate that I can do all things through Christ Jesus, who strengthens me.

Gary Alen Whitted

Table of Contents

Page

List of Figures

Figure | Page

List of Tables

Table	Page

Table | Page

Table | Page

DETERMINATION OF THE UNDERLYING TASK SCHEDULING ALGORITHM FOR AN ADA RUNTIME SYSTEM

I. Introduction

Since DoD regulations mandate the use of Ada for real-time systems development, predictable Ada task scheduling performance is important to the software developers of real-time DoD systems. Current Ada tasking rules produce task scheduling results which are unpredictable and implementation dependent. A method to identify the underlying task scheduling algorithm used by a given compiler would aid immensely in designing real-time systems with Ada. This thesis addresses the development of a suite of Ada programs to reveal, for any Ada compiler, the underlying task scheduling algorithm it uses. Research in this area should result in a compiler evaluation tool for use by software developers to allow them to determine the scheduling algorithm used by a given Ada run-time system.

1.1 Background

An Ada task is a programming entity that can be executed in parallel with other programming entities or can be considered to be executed by a logical processor of its own. Due to their unique timing requirements, "real-time systems are designed as a set of cooperating concurrent processes (Ada tasks) using the Ada tasking model" (5:49). The Ada tasking model, which includes task synchronization and rendezvous, along with the 'DELAY' statement and the 'PRIORITY' pragma, is the basic framework for real-time system design in Ada. The general requirements of Ada task scheduling, task rendezvous, and task synchronization processing are outlined in Chapter 9 of the Ada Language Reference Manual (LRM), ANSI/MIL-STD-1815A (12:Sec 9).

Specific characteristics of the underlying algorithm used to implement task scheduling on a given Ada compiler has been left to the discretion of the individual compiler vendor. Typically, the task scheduling algorithm used by the compiler vendor is proprietary and not available to real-time system designers. The efficiency of a real-time system, and the specific order in which tasks are serviced can be significantly affected by the type of scheduling algorithm used. The ambiguity in the Ada tasking requirements has led to several problems with the development and portability of real-time systems using Ada. Therefore, a method of determining the type of task scheduling algorithm used, would help Mission Critical Computer Resource (MCCR) software developers select the compiler which is best suited for their particular application.

1.2 Statement of the Problem

As noted above, the problem is to identify the task scheduling algorithm used by a given Ada run-time system. One possible method of determining a compiler's underlying task scheduling algorithm is to produce a test suite of Ada programs which, when compiled on a given compiler and run, will reveal the scheduling algorithm used by that compiler. The objective of this thesis is to determine whether it is experimentally feasible to design such a tool. If it is feasible, I will design a sample testbed of Ada programs and demonstrate the ability to identify a compiler's underlying task scheduling algorithm.

There are several difficulties associated with developing an Ada task scheduling evaluation tool. First, developing this tool may not be feasible. The question of feasibility centers around the problem of capturing the task scheduling characteristics from the run-time system using high-level programs. Next, if the extraction is feasible, determining which scheduling characteristics need to be considered and which can be extracted using Ada may be difficult. Finally, the analysis, design, and coding of any Ada program can be very difficult. The research into how an Ada run-time system schedules tasks may be examining Ada task scheduling at a much lower level than any previous research. But, the current literature clearly illustrates that the Ada tasking model has some limitations which need further investigation.

1.3 Summary of Current Knowledge

The only construct provided in Ada to specifically designate the order for task execution is the pragma PRIORITY *static expression* statement. However, this only provides control over task execution when two tasks of different priority are awaiting execution. The *static expression* is an integer representing the priority such that a lower value indicates a lower degree of urgency. The Ada LRM provides only the following rule with regard to task scheduling:

> If two tasks with different priorities are both eligible for execution and could sensibly be executed using the same physical processors and the same other processing resources, then it cannot be the case that the task with the lower priority is executing while the task with the higher priority is not. (12:Sec 9,16)

Although this rule specifies that a higher priority task will run prior to a lower priority task when they are both ready to run, there is no indication as to which task will execute first when the two tasks have equal priority or have no priority defined. Additionally, there is no indication as to whether a lower priority task which is executing should be interrupted when a task of higher priority becomes ready. Finally, there is no indication as to which rendezvous will be executed first when there are several tasks awaiting separate entry calls at an *open alternative* select statement.

As noted above, except for the PRIORITY rule, specific requirements of the underlying algorithm used to implement task scheduling on a given Ada compiler is left to the discretion of the individual compiler vendor. There are many scheduling algorithms available for the compiler vendor to choose. Since the specific order in which tasks are serviced can be significantly affected by the scheduling algorithm used, the efficiency of a real-time system developed in Ada is significantly impacted by the scheduling algorithm.

This ambiguity in the requirements for Ada task scheduling has led to several problems with the development of real-time systems using Ada. Not all of the literature focuses on the same problems, but the articles discussed later in chapter 2 identify several common problems. There are two avenues of research which can be pursued to resolve the ambiguous Ada tasking environment problem: (1) operate blindly, without any knowledge of a compiler's task scheduling algorithm, and identify problem work-arounds, or (2) detect the underlying task scheduling algorithm and select the compiler which best supports the scheduling requirements of the real-time system under development.

1.4 Assumptions

The following assumptions were made at the onset of this thesis effort. First, I assumed that specific information about the underlying task scheduling algorithm used by at least one Ada compiler would be available. With this information, I planned to validate the testbed of Ada programs for at least that one case. Without the information, I would not be assured that the testbed worked properly.

Additionally, after the initial research into the characteristics of scheduling algorithms, I decided to narrow down the scope of this detection effort. Therefore, I made the assumption that most Ada compilers probably use one of the simple, well-known scheduling algorithms (commonly used in operating systems) for task scheduling.

1.5 Scope of the Thesis Project

There are many different scheduling characteristics and parameters which can be included in a given scheduling algorithm. Initially, I tried to look at all scheduling algorithms, including those which are applicable to real-time processing. But, this included many complex algorithms which were not very likely candidates for use in Ada run-time systems. So, the scope of this project's software development effort (i.e. the testbed) was limited to simple scheduling algorithms with characteristics which are detectable by running a high-level Ada program. Therefore, to demonstrate the feasibility of this approach, the final testbed was limited to only differentiating between one of five basic scheduling algorithms.

It is clear from the literature that further research into scheduling algorithms and possible changes to the Ada rules are currently being pursued to improve Ada's real-time efficiency. Short of developing any new algorithms or changing the Ada language rules, providing a tool to identify the underlying task scheduling algorithm used by an Ada run-time system will be an asset to the real-time system designer. Successful completion of this thesis research should lay the groundwork for providing such a tool.

1.6 Standards

The Ada Language Reference Manual, MIL-STD-1815A, was used and referenced throughout this research. This standard identifies the constructs and rules of the Ada language. Also, it is the standard by which compiler vendors develop Ada compilers and by which the Ada compilers are validated.

1.7 Approach/Methodology

First, I refined the problem definition through an in-depth literature search. This research focused specifically on run-time scheduling characteristics which can be detected by a high-level language. Then, I identified a set of test cases which could be used to reveal the scheduling algorithm characteristics exhibited by a run-time system. Next, I defined and analyzed the software requirements. Then, I designed, coded, and tested the Ada programs for the testbed. Finally, I validated the testbed with three Ada compilers for which the underlying task scheduling algorithm was known.

1.8 Thesis Organization

In chapter 2 of this thesis, a detailed literature search is provided. This literature search includes a look at real-time scheduling problems with Ada and an overview of the work being done to resolve those problems. It also includes an overview of scheduling algorithm research, a discussion of scheduling algorithm characteristics, and the description of several scheduling algorithms.

In chapter 3, I have documented the analysis which I used to determine what requirements were necessary to detect the task scheduling characteristics of an Ada run-time system. The task parameters which were controled, as well as the scheduling characteristics which were measured, are also identified. Then, the test cases which incorporate different combinations of the task parameter relationships are discussed. Finally, the execution result predictions are listed and discussed.

Chapter 4 describes the design and development of the Ada programs which model the test cases. This, includes a discussion of the Ada constructs which were used and the overall structure of the parent programs. In chapter 5, I provide the results of executing the test case programs an several Ada compilers. The results of the execution on each of the five compilers is tabulated. and the analysis of these results produced from each compiler is discussed. Finally, in chapter 6, the conclusions reached as a result of this research are provided and some reccomendations for further research in this area are made.

II. Literature Review

The first part of this review addresses current research in the problems associated with the use of Ada tasking constructs. Fundamental real-time scheduling requirements, the limitations encountered with Ada providing these requirements, some methods used to investigate these limitations, and some suggested work-arounds are discussed in the first part. The second part of this review addresses task scheduling algorithm characteristics. It describes the common task scheduling parameters, the available scheduling algorithms, and the parameters which should be measurable for an Ada run-time system.

2.1 Current Research Observations of Ada Task Scheduling

2.1.1 Real-Time Scheduling Requirements. There are several fundamental requirements of a real-time programming language. To facilitate proper scheduling in real-time system design, Dennis Cornhill identified the need for an integrated approach to critical system resource management to avoid missed deadlines or underutilization of resources; a predictable scheduling algorithm; a scheduler which manages both periodic and aperiodic jobs, as well as jobs with stochastic execution times; and a preemptive scheduler (8:34-35). Douglass Locke has also pointed out that the run-time environment should utilize minimal overhead for resource allocations, have predictable response times, and have modifiable priorities (22:51-52).

In the articles noted above, the authors described why their identified requirements are essential to real-time processing and how Ada falls short of satisfying these requirements. While the identification of an Ada run-time system's task scheduling algorithm will not help satisfy all of the requirements noted by Cornhill and Locke, it will aid in predicting task scheduling and response time, determining the types of jobs which the scheduler can manage, and identifying whether the scheduler is preemptive. To design effective real-time systems, software design engineers need to know how the run-time system schedules tasks for execution. This is necessary to understand which real-time scheduling requirements are being satisfied by the run-time system, and which need to be satisfied through the application software.

2.1.2 Specific Ada Limitations. There are several limitations associated with using Ada for real-time system design. These limitations have been identified as priority inversion, nondeterministic task execution, difficult execution of preemptive scheduling, and the lack of a real-time executive. Each of these limitations are discussed below.

Priority inversion is a condition where low priority tasks are allowed to needlessly block higher priority tasks. The occurrence of priority inversion in Ada programs was identified by several authors as a significant limitation to designing real-time systems with Ada (4:8) (9:30) (19:53) (21:39). An example of this would be when there is a server task of priority P_S servicing a set of consumer tasks with priorities P_L through P_H, where P_L is the lowest consumer priority and P_H is the highest. In the cases where $P_S = P_L or P_S = P_H$, the servicing of consumer-server rendezvous would be nondeterministic due to Ada's priority rule. Therefore, the two cases where $P_S < P_L$ or $P_S > P_H$ are used to illustrate priority inversion. In the first case, if the server task is not ready to accept the request and there are other consumer tasks ready to execute, a high priority consumer task may be blocked while calling the server task. In the second case, a high priority consumer task which has just become eligible for execution may be preempted by the server task which is doing work for a low priority consumer task (9:31). In both cases, the task which is started by the scheduler may not be the one with the highest priority of the tasks which are ready to run.

Dennis Cornhill suggested the priority inheritance scheme as a work-around to prevent priority inversion. With this scheme the priority of the clients waiting for service is passed on to (or inherited by) the server task. In this way, "priority inversion can be avoided if the server always selects for service the highest priority waiting client and inherits its priority from its waiting clients as well"(9:32). But, in Ada, since there is only a single level of priority passing during a rendezvous, the server only inherits the priority of the first-level client if that client's priority is higher than that of the server. The server doesn't inherit the priority of any second-level clients which are waiting (9:32). Thus, the rules of an Ada rendezvous illustrate that Ada's limited form of priority inheritance is not adequate to prevent priority inversion.

The authors of three separate articles revealed that Ada exhibits nondeterministic task execution behavior because of the way it handles *open alternative* select statements and because of its First In/First Out (FIFO) entry call queuing (4:8) (21:39) (23:49). Since there are separate queues associated with each entry call, there are several queues associated with an open alternative select statement. The priority of each queue corresponds to either the priority of the task in which the entry call is located, or the priority of the calling task, whichever is higher. The queuing of individual entry calls is FIFO when all tasks are allocated with the same priority, but the selection of which queue to service first at an open alternative select statement is not specified. Thus, when all tasks are allocated with the same priority or without any priority, the results are unspecified by the Ada LRM and implementation-dependent (1:43) (8:34). This results in the system designer's lack of control over the execution of several time critical tasks.

Another problem, somewhat related to the priority inversion problem, is the difficulty in executing preemptive scheduling within the Ada run-time environment. Since Ada requires all instances of the same task type to have the same priority, and that priority cannot change dynamically, preemptive scheduling is difficult without the costly overhead of special priority passing paradigms (5:50) (1:43,45). In some Ada run-time systems, a Round-Robin (time-sliced) algorithm may actually be employed for scheduling task execution. Thus, a given Ada run-time systems may already incorporate a preemptive algorithm at the lowest level of task scheduling. If this type of scheduling algorithm could be detected, other types of more sophisticated preemptive scheduling may be possible at the overall system level.

In *An Assessment of the Overhead Associated with Tasking Facilities and Task Paradigms in Ada*, Thomas Burger identified what he considers to be Ada's key limitation by stating:

> Since Ada does not include a real-time executive, task activation and termination are not accomplished via programmer written executive service requests. Task activation and termination in Ada is a part of the tasking model semantics, and is performed automatically based on an elaborate set of rules. (5:51)

The task scheduling portion of this 'elaborate set of rules' consists of the single rule noted above, and the ambiguity of this rule has already been discussed. Without a real-time executive, the real-time system developer must try to simulate a real-time environment using the components of the Ada tasking model. Using the Ada tasking model forces the developer to rely on the Ada run-time system to schedule task execution.

Since Ada's task scheduling rule is so ambiguous and the timing overhead associated with tasking is so excessive, an efficient real-time system cannot be designed using Ada as it is currently defined. This problem may be overcome if the designer can identify and understand the task scheduling algorithm being used by several Ada run-time systems, then select the appropriate Ada compiler and design the real-time system accordingly. Additionally, if the underlying task scheduling algorithm is known, the designer may be able to avoid priority inversion, eliminate nondeterministic task execution, and more readily design preemptive scheduling systems in Ada. Since Ada's rules for task scheduling are so ambiguous, an efficient real-time system cannot be designed using Ada unless the designer can identify and understand the task scheduling algorithm being used.

2.1.3 Previous Attemps to Solve Ada's Limitations. In 1987, Dennis Cornhill identified a stabilized rate monotonic algorithm as a potential way of facilitating real-time system design, but pointed out that current Ada rules prevent the use of this type of algorithm. With this algorithm "certain high priority tasks run for limited periods only. When this time period elapses, if the execution has not been completed, the job must be preempted by lower priority jobs for another well defined length of time" (8:34). Thus, the stabilized rate monotonic algorithm uses information about job importance, periodicity, and average and worst case execution times for scheduling decisions. In order to permit use of hard deadline scheduling algorithms (i.e. the stabilized rate monotonic algorithm), Cornhill concluded that two areas of Ada need to be changed. First, all run-time scheduling operations should consider a task's priority. Second, "constraints on the definitions for priority and the language's scheduling policy should be relaxed" (8:35-37). Once again, if the underlying task scheduling algorithm is known, designing hard deadline scheduling systems with Ada may be feasible without changing the language. However, software which is based on a particular algorithm may not be portable.

While leading the *Tasking* session at the 1987 ACM International Workshop on Real-Time Ada Issues, Cornhill summarized the session with several recommendations. First, he suggested that deadline scheduling problems be addressed in the 9X revision to Ada. Next, he suggested that a clarification be issued by the Ada Language Maintenance Panel to eliminate the synchronization point of the 'ACCEPT' statement for an interrupt. And finally, he noted that built-in priority management packages should be provided by compiler vendors (7:32).

At the same workshop, Gary Frankel identified four special concurrency paradigms (monitor/process structure, asynchronous message passing, interrupt procedures, and event signaling) to make Ada tasking useful. Using these special case paradigms, Frankel claimed that "Ada tasking can be made as efficient as any other method of concurrency programming" (14:47-48).

Several proposed environments were presented at the 1988 ACM International Workshop on Real-Time Ada Issues and published in *Ada Letters, 1988 Special Edition*. In *A Testbed for Investigating Real-Time Ada Issues*, Mark Borger discussed the Software Engineering Institute's (SEI) 'Ada Embedded Systems Testbed' project which they used to "provide a real-time laboratory environment for conducting experiments using Ada and investigating real-time Ada issues" (4:7). Using this testbed, researchers at SEI investigated some promising real-time scheduling algorithms that were developed to overcome Ada's aperiodic task servicing problem. Specifically, they looked at the rate monotonic algorithm, a priority inheritance based scheduling algorithm, and a deferrable server algorithm. Implementation of these algorithms revealed that researchers need to look at solutions which are either "constrained by current Ada implementations" or "involve legal extensions or allowable interpretations of the language semantics" (4:7-10). Although the researchers at SEI looked at high-level scheduling algorithms, it may be possible to use a similar testbed to identify the low-level task scheduling algorithm used by an Ada run-time system.

Two other articles presented at the 1988 ACM Workshop addressed the rate monotonic algorithm. In his article, John B. Goodenough, another researcher at SEI, showed that the basic priority inheritance and priority ceiling protocols (both rate monotonic algorithms) corrected Ada's unbounded priority inversion problem. Although the priority ceiling protocol seemed to perform well, researchers at SEI are trying to extend the protocol and verify its utility (16:24). In a separate article, Douglass Locke described an experiment using the rate monotonic algorithm in a modified Ada run-time environment which "confirmed earlier theoretical analysis that priority inheritance can provide substantial benefits" (21:40-42).

In work totally unrelated to the 1988 ACM Workshop, Jane W.S. Liu proposed an Imprecise Computation Approach to improve on the rate monotonic algorithm's schedulability and processor utilization during fluctuating system loads. Liu suggested that the deficiencies in Ada's existing priority mechanism could be corrected by introducing data structures (i.e. tables of repetition rates and deadlines) at link-time which cooperate with the run-time system and have little impact on the existing language definition (20:33-34).

The discussion above reveals that most of the research done thus far has focused on Ada's limitations with respect to real-time design, on work-arounds to use Ada for real-time design, and on suggestions for changing Ada to improve it's real-time capabilities. These work-arounds appear in the form of high-level algorithms implemented using the Ada tasking model. But, I have found no evidence of research in the area of task scheduling algorithm detection for Ada run-time systems. Since there are many possible algorithms which could be used by an Ada run-time system for task scheduling, detection of the specific scheduling algorithm may be impossible if the scope of detection includes all possible algorithms. But, it seems reasonable to narrow this scope to a few simple, well-known algorithms which are the most likely to be used by an Ada compiler. With this restriction, it may be possible select the Ada compiler which exhibits characteristics of the scheduling algorithm most appropriate for a given real-time application. That concept is the thrust of this research effort.

2.2 Scheduling Algorithm Detection Research

According to Coffman and Kleinrock, "the goal of scheduling algorithms is to provide the population of users with a high grade of service (rapid response, resource availability, etc.) at the same time maintaining an acceptable throughput rate" (6:11). Although their statement was made with regard to computer scheduling in general, it also applies specifically to task scheduling within an Ada run-time system. However, within an Ada run-time system, the users are represented by the individual tasks awaiting execution and an acceptable throughput rate is achieved when all tasks are serviced in a manner such that all deadlines are met.

When designing an Ada program to support real-time applications, the designer has to be concerned with *time-critical processes* (TCPs). According to Omri Serlin, TCPs are "computational procedures bound by ***hard deadlines***", such that failure to meet the deadline "results in an irreparable damage to the computation" (24:925). When dealing with real-time applications, an efficient scheduling algorithm is one that "guarantees to each TCP sufficient processor time to complete its task before its deadline, while minimizing *forced* idle CPU time" (24:925). Scheduling algorithms of this nature are called Hard-Deadline scheduling algorithms. Typically, Hard-Deadline algorithms are much more complex than standard scheduling algorithms and are not likely to be implemented as an Ada run-time system's algorithm. When such an algorithm is required for a real-time system, it is implemented on top of the Ada run-time systems using special tasking paradigms.

In general, an Ada compiler may be required to produce a run-time system which controls the scheduling of tasks on multiple processors. However, this is only the case when there are parallel processors available in the hardware architecture. Though this is becoming more commonplace in large and medium scale computer systems, it is not the case with small and embedded computer systems. Most DoD embedded real-time systems have been designed on single processor based architectures (i.e. M68000, Z8000, and MIL-STD-1750A). Therefore, the following discussion will focus on scheduling algorithms for single processor systems without hard deadlines.

In his book, *An Introduction to Operation Systems*, Harvey Deitel identifies the general objectives of a scheduling algorithm as:

- Provide fair treatment to all waiting processes (or tasks),
- Maximize CPU throughput,
- Provide predictable response,
- Reduce process scheduling overhead,
- Balance system resource utilization,
- Provide a reasonable balance between system response and utilization,
- Avoid process starvation (or indefinite postponement),
- Acknowledge process priorities, and
- Provide graceful degradation (10:250-251).

These objectives are equally applicable to Ada run-time systems. The detection of an Ada run-time system's scheduling algorithm is primarily concerned with the degree to which the algorithm satisfies the 'predictable response' objective. If a given algorithm has a predictable response which distinguishes it from other algorithms, then the execution of a predetermined set of tasks can be observed and analyzed to detect the algorithm used.

2.2.1 Scheduling Algorithm Characteristics. Although the successful achievement of some objectives noted above may be very subjective, the achievement of others can be measured by looking at certain characteristics. First, to be consistent throughout the remainder of this thesis, the term 'task' will be used in lieu of the terms 'process' or 'job'. The following scheduling characteristics are typically measured (for a given task i) to compare algorithm performance:

- Arrival time (A_i) [the time when the task initially arrives and is ready to execute],
- Start time (S_i) [the time when the task actually begins execution],
- Finish time (F_i) [the time when the task actually finishes execution],
- Service time required (C_i) [the actual CPU service time required for the continuous execution of a task without interruption],
- CPU burst time (β_i) [the continuous burst of CPU service time required between I/O requests or other interrupts, $\sum(\beta_i) = C_i$],
- CPU utilization [for an algorithm processing n tasks] ($Totaltime / \sum_{j=1}^{n} C_j$).
- System throughput [Number of tasks processed per unit time] ($n/Totaltime$),
- Process turnaround time ($F_i - A_i$),
- Process response (or completion) time ($T_i = F_i - S_i$),
- Process waiting time ($W_i) = T_i - C_i$),
- Penalty ratio ($P = t/C_i$, where "t is time in execution before task i can leave the ready list because it will either finish or will need to wait for something "(13:17)), and
- Response ratio ($R = C_i/t$ (13:17)).

The application of a specific scheduling algorithm to a given set of tasks should produce a set of measurable characteristics. Although not necessarily unique for a single set of tasks, application of the algorithm to a selected suite of task combinations may produce a set of characteristic measurements which are unique for that algorithm.

2.2.2 Scheduling Algorithm Descriptions. Prior to identifying the requirements for a suite of test cases to detect the scheduling algorithm used by a given Ada run-time system, the scheduling algorithms most likely used in Ada run-time systems will be identified. As noted earlier, due to the complexity and costly overhead associated with ***hard-deadline*** algorithms, these will not be considered as candidates. Additionally, based on the assumption that most DoD embedded systems are single-processor architectures, multiprocessor scheduling algorithms also will not be considered. Thus, the focus of this research will be on simple, well-known, single-processor scheduling algorithms such as First-Come-First-Serve (FCFS), Round-Robin (RR), Shortest-Job-First (SJF), Priority, and Highest-Penalty-Ratio-Next (HPRN), and Multi-Level Feedback. Only those which have a high potential of being implemented as part of an Ada run-time system will be checked for.

2.2.2.1 First-Come-First-Serve (FCFS). This scheduling algorithm is characterized by the First-In-First-Out (FIFO) serving queue. Tasks are lined up in a ready queue as they arrive. This is the simplest scheduling algorithm to write and understand. However, its performance is often quite poor. The average waiting time is generally not minimal, as a shorter task may have to wait quite some time before execution if a longer task arrives first. Additionally, the average waiting time may vary substantially depending on the sequence of tasks awaiting execution. This is a non-preemptive algorithm; thus, once a task is started, it will run to completion or until it is blocked (i.e. due to an I/O request, a delay, or a rendezvous).

While the Ada LRM specifies that each entry call queue is required to process calls in the order of arrival (i.e. a FIFO queue) (12:Sec 9,9), there is no such requirement for scheduling task execution. Although not the most efficient in terms of average waiting time; due to its simplicity, this may be the algorithm of choice for some Ada compiler vendors.

2.2.2.2 Round-Robin (RR). A scheduling algorithm in which each task is allocated a slice of execution time on the CPU is called Round-Robin (RR). In this algorithm, as tasks arrive they are placed on a ready queue in a FIFO fashion. But, when they get to the front of the queue, they are only permitted a limited time for execution. If they complete within that time, they exit the queue; however, if they block for I/O or need more CPU time they are interrupted and placed at the back of the ready queue. Thus, RR is a preemptive algorithm where the ready queue is treated as a circular queue, and each uncompleted task has a short turn at execution each time the scheduler cycles through the queue. A shorter task may complete during the time slice and exit, whereas a longer task may require several trips through the queue before completion. The performance of this algorithm depends on the designated time slice. To function efficiently, approximately eighty percent (80%) of the cpu bursts should be shorter than the designated time slice. If the time slice is too large, RR degenerates into FCFS because each task completes within one time slice. And if the time slice is too small, the context switching overhead swamps the CPU (25:166-168).

As noted earlier, with regard to task execution, the Ada LRM states:

> The execution of a program that does not contain a task is defined in terms of a sequential execution of its actions, ... These actions can be considered to be executed by a single *logical processor*. Tasks are entities whose executions proceed *in parallel* in the following sense. Each task can be considered to be executed by a logical processor of its own. Different tasks (different logical processors) proceed independently, except at points where they synchronize. (12:Sec 9,1)

Considering the requirement noted above, in order for an Ada run-time system to execute tasks in parallel (or give the impression of that more than one processor was being used), some type of RR scheduling algorithm would seem most appropriate on a single processor system. This is particularly brought out by the Ada LRM statement that "parallel tasks (parallel logical processors) may be implemented on multicomputers, multiprocessors, or with interleaved execution on a single *physical processor*" (12:Sec 9,1). Therefore, RR should have a high probability of use in Ada run-time systems.

2.2.2.3 Shortest-Job-First (SJF). There are two versions of the SJF algorithm, a static version and a dynamic version. The static SJF algorithm requires some prior knowledge of a task's projected CPU service time C_i requirement. Using this information, the static SJF algorithm sorts a task set based on increasing C_i and executes them in that order. The static SJF algorithm would most likely be used for (long-term) scheduling in a batch environment where task service time requirements are known prior to execution.

On the other hand, the dynamic SJF algorithm requires no prior knowledge of C_i requirements. This version of SJF puts new task arrivals at the front of the ready queue to execute as soon as the currently running task is blocked. When its turn comes up, the task is permitted to run until a block occurs for an I/O request, a rendezvous, a delay, or some other task generated reason. After the task is blocked, but prior to placing it back on the ready queue, the scheduler projects the next CPU burst β_{n+1} requirement based on the most recent CPU burst time β_n used prior to blocking. After each cycle through the ready queue, the algorithm sorts the remaining tasks based on their projected β_{n+1} and executes the task with the shortest β_{n+1} first. In this manner, tasks which are I/O intensive and only require small bursts of CPU processing are given priority over CPU intensive tasks.

With either version of SJF, once a task is started it will run until it requests a block or until it is finished. Thus, both the static and dynamic SJF algorithms are non-preemptive. This results in a minimum average waiting time for a given set of tasks. However, SJF requires either prior knowledge of a task's required service time (i.e. C_i) or the additional overhead associated with predicting the next CPU burst (i.e. β_{n+1}).

It's not very likely that an Ada run-time system would be using either of these versions of SJF for two reasons. First, there's no other requirement to provide any projection of a task's expected

C_i. And second, the overhead associated with predicting all of the tasks' β_{n+1} based on their most recent β_n could be extremely high. But, it should be easy to detect because of the algorithm's characteristic minimum average waiting time.

2.2.2.4 Priority. In this scheduling algorithm, each task has a priority associated with it. When the tasks queue up awaiting execution, the tasks with the highest priority are always placed at the head of the queue. Thus, the tasks with the highest priority are executed first. Tasks of equal priority are scheduled using some default algorithm. The Priority algorithm is also non-preemptive. The major problem with this algorithm is the possibility of indefinite blocking or starvation of lower priority tasks where they don't get an opportunity to execute.

The rules of Ada dictate that some level of Priority scheduling must be used when tasks have a PRIORITY assigned. But, still there is no requirement that any specific algorithm be used to schedule tasks with equal priority. Thus, a Priority algorithm which degenerates to some default algorithm should be a prime candidate for use in Ada run-time systems.

2.2.2.5 Highest-Penalty-Ratio-Next (HPRN). Under the category of non-preemptive scheduling algorithms, either long tasks are given an unfair advantage under the FCFS algorithm or short tasks are given an unfair advantage under the SJF algorithm. According to Finkel, by calculating a 'penalty ratio' and selecting the task with the highest penalty ratio for the next execution, the scheduling of tasks becomes 'fairer' (13:24). The penalty ratio is calculated by dividing the response time, T_i, (i.e. $F_i - S_i$) by t, where t is the time in execution before a task can leave the ready list. According to Harvey Deitel, this amounts to assigning dynamic priorities to the tasks based on the calculated penalty ratio (10:258). The disadvantage of this algorithm is that it is more expensive to implement due to the required calculation of the penalty ratio for all tasks prior to executing a task. Additionally, a short task arriving immediately after a long task has begun execution will still have to wait to start. It's very unlikely that this algorithm is used in Ada run-time systems because priorities are static and the overhead may be too costly.

2.2.2.6 Multi-Level Feedback Scheme. This algorithm employs several queues for tasks which are awaiting execution. The algorithm is defined by the number of individual queues, the scheduling algorithm for each queue, the criteria required for a task to move from one queue to the next higher queue, the criteria to move a task to the next lower priority queue, and the initial assignment criteria. Each queue has a different priority and the queue in which a task is placed is determined by the cause of the most recent execution interrupt. Any newly arriving task is allowed to preempt existing tasks until it has been given an amount of CPU time equivalent to that used by existing tasks. The multi-level feedback algorithm is an adaptive mechanism which responds to changes in tasking requirements, but requires considerable overhead to operate effectively. Thus, this type of algorithm is very unlikely to be implemented in Ada run-time systems (10:259-261)(13:24-25)

2.3 Summary

This review has provided some basic scheduling algorithm information which will be used to identify the requirements for the investigation of Ada task scheduling. Current research into real-time scheduling requirements and the limitations associated with using Ada for the development of real-time systems reveal the need for changes to Ada tasking rules. With the current ambiguous Ada tasking rules, different implementations of Ada may produce different results. It is clear from the literature that further research into scheduling algorithms and possible changes to the Ada rules are required to improve Ada's real-time efficiency. The current literature also reveals that some methods are being investigated to overcome the Ada limitations mentioned. Short of developing any new work-arounds or changing the language rules, an alternate approach might be to determine the task scheduling algorithms used by a set of available Ada compilers, and then select the compiler which is best suited for the job at hand. In support of this approach, the literature review provided a discussion of scheduling algorithm characteristics. This provides the background for the possible development of a testbed of Ada programs to detect the underlying task scheduling characteristics exhibited by a given Ada compiler. Based on the information provided, and in order to limit the scope of the development effort, the test suite will only check for the FCFS, RR, Static SJF, Dynamic SJF, and Priority algorithms. The next chapter provides a discussion of the requirements analysis used to develop the test suite of Ada programs.

III. Requirements Analysis for Ada Task Scheduling Detection

The detection of the scheduling algorithm used by a run-time system will require the measurement of one or more algorithm characteristics to distinguish among the five algorithms. There are several approaches which are used to predict algorithm performance. When the task parameters are dynamic, queuing models are used to predict the performance of scheduling algorithms. Several authors have used queuing theory to evaluate scheduling algorithm performance on dynamic task sets (18, 17, 24). On the other hand, when the task parameters are static, an evaluation method known as deterministic modeling can be used. Several authors have used flow-time analysis and Gantt charts to predict the sequence of task execution for a given scheduling algorithm known apriori (15, 13, 25). The scheduling methods defined in Chapter II can be described using the deterministic approach, therefore the basis for the development of the Ada testbed will be the same.

This requirements analysis will discuss the task parameters and scheduling algorithm characteristics which can be used to distinguish among the five algorithms under investigation, and the expected results for test cases which are used to model different parameter relationships.

3.1 Scheduling Algorithm Characteristics/Parameters

In *Deterministic Processor Scheduling*, M.J. Gonzalez, Jr. used Gantt charts and flow-time measurement to analyze several single-processor algorithms (15:179–181). In his book, *An Operating Systems Vade Mecum*, Raphael Finkel used Gantt charts along with known task parameters to illustrate and compare the results of applying various algorithms to a given set of tasks (13:20–27). With respect to task scheduling analysis, a Gantt chart is a tabular representation of task execution during a sequence of predetermined time increments. Flow-time analysis is concerned with the sequence of, and relationship between, the start and finish times of the tasks. Since the execution sequence for a given task set will vary depending on which scheduling algorithm is used and how the parameters of the tasks are related, representation of the expected execution results using a Gantt chart requires prior knowledge of one or more task parameters.

Initially, I believed that the expected Gantt chart produced by various algorithms for a given task set could be predicted if the arrival time (A_i), service time (C_i), and priority (P_i) of the tasks were known. Although many examples of Gantt chart analysis contain several tasks within the given task set, algorithm detection may be possible with only two tasks in the task set. But, in order to do it with only two tasks, (A & B), all the possible equality relationships between A_A & A_B, C_A & C_B, and P_A & P_B for the two tasks had to be observed. Twenty-seven test cases cover each combination of these parameter relationships for two tasks. A listing of the test cases, along with the corresponding parameter relationships for the two tasks, is provided in Table 3.1.

Originally, I thought that these test cases would be sufficient to detect RR, FCFS, SJF, and Priority. Later, I realized that there was a distinction between the results predicted for the Dynamic SJF algorithm and the Static SJF algorithm. This distinction, along with other problems, resulted in the need for another special test case which will be discussed later. It wasn't until after the Ada programs which modeled these twenty-seven test cases were executed, and the results of the execution analyzed, that I discovered these test cases could not detect a Dynamic SJF algorithm. Thus, the term SJF will be used to refer to the Static SJF algorithm until the special test case is presented.

The A_i, C_i, and P_i task parameters were selected for use in the test cases for the following reasons. First, if the task arrival times for two tasks are known in advance, this knowledge can be used for detecting a FCFS algorithm. Since a FCFS algorithm executes the task with the earlier arrival time prior to the other task, the resulting execution sequence and the start & finish times can be predicted.

If the task service times are also known in advance, a better approximation for the expected start and finish times is also possible. In order to more accurately predict start and finish times, the

Scheduling Algorithm Detection Test Cases			
Test Case	*Service Time*	*Arrival Time*	*Priority*
1	$C_A = C_B$	$A_A = A_B$	$P_A = P_B$
2	$C_A = C_B$	$A_A = A_B$	$P_A > P_B$
3	$C_A = C_B$	$A_A = A_B$	$P_A < P_B$
4	$C_A = C_B$	$A_A < A_B$	$P_A = P_B$
5	$C_A = C_B$	$A_A < A_B$	$P_A > P_B$
6	$C_A = C_B$	$A_A < A_B$	$P_A < P_B$
7	$C_A = C_B$	$A_A > A_B$	$P_A = P_B$
8	$C_A = C_B$	$A_A > A_B$	$P_A > P_B$
9	$C_A = C_B$	$A_A > A_B$	$P_A < P_B$
10	$C_A > C_B$	$A_A = A_B$	$P_A = P_B$
11	$C_A > C_B$	$A_A = A_B$	$P_A > P_B$
12	$C_A > C_B$	$A_A = A_B$	$P_A < P_B$
13	$C_A > C_B$	$A_A < A_B$	$P_A = P_B$
14	$C_A > C_B$	$A_A < A_B$	$P_A > P_B$
15	$C_A > C_B$	$A_A < A_B$	$P_A < P_B$
16	$C_A > C_B$	$A_A > A_B$	$P_A = P_B$
17	$C_A > C_B$	$A_A > A_B$	$P_A > P_B$
18	$C_A > C_B$	$A_A > A_B$	$P_A < P_B$
19	$C_A < C_B$	$A_A = A_B$	$P_A = P_B$
20	$C_A < C_B$	$A_A = A_B$	$P_A > P_B$
21	$C_A < C_B$	$A_A = A_B$	$P_A < P_B$
22	$C_A < C_B$	$A_A < A_B$	$P_A = P_B$
23	$C_A < C_B$	$A_A < A_B$	$P_A > P_B$
24	$C_A < C_B$	$A_A < A_B$	$P_A < P_B$
25	$C_A < C_B$	$A_A > A_B$	$P_A = P_B$
26	$C_A < C_B$	$A_A > A_B$	$P_A > P_B$
27	$C_A < C_B$	$A_A > A_B$	$P_A < P_B$
where C is the service time, A is the arrival time, and P is the priority.			

Table 3.1. Algorithm Detection Parameter Relationships for Test Cases 1 thru 27

service time inequality relationships identified in Table 3.1 were converted to equality relationships. The relationship $2C_A = C_B$ was used to obtain the $C_A < C_B$ relationship, and $C_A = 2C_B$ was used to obtain the $C_A > C_B$ relationship. The doubling of service times for the equality relationship was arbitrarily selected to simplify the start and finish time predictions (and analysis). The prior knowledge of task service times also aids in the detection of the SJF algorithm. Since a SJF algorithm executes the task with the shorter service time prior to the other task, the resulting execution sequence can be identified and a close approximation to start and finish times can be predicted.

If the priorities of two tasks are known in advance, detection of the Priority algorithm should be possible. Since a Priority algorithm executes the task with the higher priority before the other task, the resulting execution sequence can be identified and a close approximation to the start and finish times can be predicted.

Finally, if a RR algorithm is used for task scheduling, the two tasks will take turns at execution. Once again, if the arrival times and service times of the two tasks are known in advance, execution sequence and a fair approximation of start and finish times can be predicted. Although accurate predictions for start and finish times are not possible without prior knowledge of the time slice (TS) used by the RR algorithm, the relationships between the task start and finish times is possible.

Even though some of the test cases were functionally equivalent to each other (just a renaming of tasks), they were kept for purposes of cross-checking their expected re ..s. I also realized that some of the test cases could produce the same results for two or more algorithms. These test cases were kept because the results of two such test cases may be intersected to single out which of the algorithms under investigation is possibly being used. For example, if the results of one test case indicates that either FCFS or SJF was used, and the results of another test case indicates that either FCFS or Priority was used; then the intersection of these results reveals that FCFS was used. Therefore, all of the test case were kept to maintain a more complete test suite.

Since the overall objective was to analyze the execution results of Ada programs which modeled these test cases, the next step was to predict the execution results of each test case executing under the RR, FCFS, SJF, and Priority algorithms. The prediction of expected results for the test cases is discussed in the next section.

3.2 Predicted Execution Results for the Test Cases

The expected result of executing a given test case under a known scheduling algorithm can be described by a characteristic set of start (S_A and S_B) and finish (F_A and F_B) times, along with a corresponding Gantt chart. As noted earlier, a Gantt chart can be used to predict the expected execution sequence for a set of tasks running under a given scheduling algorithm. The Gantt charts for each of the twenty-seven test cases, executing under four scheduling algorithms, are shown in Tables A.1 through A.30 of Appendix A. The four algorithms represented are Round Robin (RR), First-Come-First-Serve (FCFS), Shortest-Job-First (SJF) [actually static SJF], and Priority.

Ideally, a single test case should have expected results which are unique for each scheduling algorithm. Examination of the Gantt charts provided in Tables A.1 through A.30 of Appendix A indicates that none of the test cases appear to be ideal in this respect. The Gantt charts are only of limited use because they do not accurately reflect the time segments, but they do provide some insight into which algorithms are detectable by a given test case. The supplemental flow-time analysis will provided additional insight when presented later. The following discussion highlights some of the observations which can be made from the Gantt charts. The predicted Gantt charts for test cases 1, 4, 7, 10, 16, 19, 22, and 25 indicate that these test cases should be useful in distinguishing RR from the other algorithms. The predicted Gantt charts for test cases 6, 8, 15, 17, 24, and 26 reveal two sets of execution sequences. One sequence indicates that either RR or Priority was used for scheduling, while the other sequence reveals that either FCFS or SJF was used. Further flow-time analysis should distinguish between RR and Priority, but not necessarily between FCFS and SJF. All other Gantt charts contain predicted execution sequences which cannot distinguish between any of the four algorithms.

The flow-time analysis produces the expected start and finish times for the execution of each test case under the four algorithms. The predicted start and finish times for each of the twenty-seven test cases are summarized in Tables 3.2, 3.3, and 3.4.

Test Case	Parameters	RR	$FCFS$	SJF	$Priority$
1	$C_A = C_B$ $A_A = A_B$ $P_A = P_B$	$S_A = 0$ or TS $S_B = TS$ or 0 $F_A = 2C - TS$ or $2C$ $F_B = 2C$ or $2C - TS$	$S_A = 0$ or C $S_B = C$ or 0 $F_A = C$ or $2C$ $F_B = 2C$ or C	$S_A = 0$ or C $S_B = C$ or 0 $F_A = C$ or $2C$ $F_B = 2C$ or C	$S_A = 0$ or C $S_B = C$ or 0 $F_A = C$ or $2C$ $F_B = 2C$ or C
2	$C_A = C_B$ $A_A = A_B$ $P_A > P_B$	$S_A = 0$ $S_B = C$ $F_A = C$ $F_B = 2C$	$S_A = 0$ $S_B = C$ $F_A = C$ $F_B = 2C$	$S_A = 0$ $S_B = C$ $F_A = C$ $F_B = 2C$	$S_A = 0$ $S_B = C$ $F_A = C$ $F_B = 2C$
3	$C_A = C_B$ $A_A = A_B$ $P_A < P_B$	$S_A = C$ $S_B = 0$ $F_A = 2C$ $F_B = C$	$S_A = C$ $S_B = 0$ $F_A = 2C$ $F_B = C$	$S_A = C$ $S_B = 0$ $F_A = 2C$ $F_B = C$	$S_A = C$ $S_B = 0$ $F_A = 2C$ $F_B = C$
4	$C_A = C_B$ $A_A < A_B$ $P_A = P_B$	$S_A = 0$ $S_B = TS$ $F_A = 2C - TS$ $F_B = 2C$	$S_A = 0$ $S_B = C$ $F_A = C$ $F_B = 2C$	$S_A = 0$ $S_B = C$ $F_A = C$ $F_B = 2C$	$S_A = 0$ $S_B = C$ $F_A = C$ $F_B = 2C$
5	$C_A = C_B$ $A_A < A_B$ $P_A > P_B$	$S_A = 0$ $S_B = C$ $F_A = C$ $F_B = 2C$	$S_A = 0$ $S_B = C$ $F_A = C$ $F_B = 2C$	$S_A = 0$ $S_B = C$ $F_A = C$ $F_B = 2C$	$S_A = 0$ $S_B = C$ $F_A = C$ $F_B = 2C$
6	$C_A = C_B$ $A_A < A_B$ $P_A < P_B$	$S_A = 0$ $S_B = TS$ $F_A = 2C$ $F_B = C + TS$	$S_A = 0$ $S_B = C$ $F_A = C$ $F_B = 2C$	$S_A = 0$ $S_B = C$ $F_A = C$ $F_B = 2C$	$S_A = 0$ $S_B = A_B$ $F_A = 2C$ $F_B = C + S_B$
7	$C_A = C_B$ $A_A > A_B$ $P_A = P_B$	$S_A = TS$ $S_B = 0$ $F_A = 2C$ $F_B = 2C - TS$	$S_A = C$ $S_B = 0$ $F_A = 2C$ $F_B = C$	$S_A = C$ $S_B = 0$ $F_A = 2C$ $F_B = C$	$S_A = C$ $S_B = 0$ $F_A = 2C$ $F_B = C$
8	$C_A = C_B$ $A_A > A_B$ $P_A > P_B$	$S_A = TS$ $S_B = 0$ $F_A = C + TS$ $F_B = 2C$	$S_A = C$ $S_B = 0$ $F_A = 2C$ $F_B = C$	$S_A = C$ $S_B = 0$ $F_A = 2C$ $F_B = C$	$S_A = A_A$ $S_B = 0$ $F_A = C + A_A$ $F_B = 2C$
9	$C_A = C_B$ $A_A > A_B$ $P_A < P_B$	$S_A = C$ $S_B = 0$ $F_A = 2C$ $F_B = C$	$S_A = C$ $S_B = 0$ $F_A = 2C$ $F_B = C$	$S_A = C$ $S_B = 0$ $F_A = 2C$ $F_B = C$	$S_A = C$ $S_B = 0$ $F_A = 2C$ $F_B = C$
where C is service time, A is arrival time, P is priority, S is start time, F is finish time, and TS is the Time Slice if RR is used,					

Table 3.2. Predicted Execution Results for Test Cases 1 - 9

Test Case	Parameters	RR	$FCFS$	SJF	$Priority$
10	$C_A = 2C_B$ $A_A = A_B$ $P_A = P_B$	$S_A = 0$ or TS $S_B = TS$ or 0 $F_A = 3C$ or $3C$ $F_B = 2C$ or $2C - TS$	$S_A = 0$ or C $S_B = 2C$ or 0 $F_A = 2C$ or $3C$ $F_B = 3C$ or C	$S_A = C$ $S_B = 0$ $F_A = 3C$ $F_B = C$	$S_A = 0$ or C $S_B = 2C$ or 0 $F_A = 2C$ or $3C$ $F_B = 3C$ or C
11	$C_A = 2C_B$ $A_A = A_B$ $P_A > P_B$	$S_A = 0$ $S_B = 2C$ $F_A = 2C$ $F_B = 3C$	$S_A = 0$ $S_B = 2C$ $F_A = 2C$ $F_B = 3C$	$S_A = 0$ $S_B = 2C$ $F_A = 2C$ $F_B = 3C$	$S_A = 0$ $S_B = 2C$ $F_A = 2C$ $F_B = 3C$
12	$C_A = 2C_B$ $A_A = A_B$ $P_A < P_B$	$S_A = C$ $S_B = 0$ $F_A = 3C$ $F_B = C$	$S_A = C$ $S_B = 0$ $F_A = 3C$ $F_B = C$	$S_A = C$ $S_B = 0$ $F_A = 3C$ $F_B = C$	$S_A = C$ $S_B = 0$ $F_A = 3C$ $F_B = C$
13	$C_A = 2C_B$ $A_A < A_B$ $P_A = P_B$	$S_A = 0$ $S_B = TS$ $F_A = 3C$ $F_B = 2C$	$S_A = 0$ $S_B = 2C$ $F_A = 2C$ $F_B = 3C$	$S_A = 0$ $S_B = 2C$ $F_A = 2C$ $F_B = 3C$	$S_A = 0$ $S_B = 2C$ $F_A = 2C$ $F_B = 3C$
14	$C_A = 2C_B$ $A_A < A_B$ $P_A > P_B$	$S_A = 0$ $S_B = 2C$ $F_A = 2C$ $F_B = 3C$	$S_A = 0$ $S_B = 2C$ $F_A = 2C$ $F_B = 3C$	$S_A = 0$ $S_B = 2C$ $F_A = 2C$ $F_B = 3C$	$S_A = 0$ $S_B = 2C$ $F_A = 2C$ $F_B = 3C$
15	$C_A = 2C_B$ $A_A < A_B$ $P_A < P_B$	$S_A = 0$ $S_B = TS$ $F_A = 3C$ $F_B = C + TS$	$S_A = 0$ $S_B = 2C$ $F_A = 2C$ $F_B = 3C$	$S_A = 0$ $S_B = 2C$ $F_A = 2C$ $F_B = 3C$	$S_A = 0$ $S_B = A_B$ $F_A = 3C$ $F_B = C + S_B$
16	$C_A = 2C_B$ $A_A > A_B$ $P_A = P_B$	$S_A = TS$ $S_B = 0$ $F_A = 3C$ $F_B = 2C - TS$	$S_A = C$ $S_B = 0$ $F_A = 3C$ $F_B = C$	$S_A = C$ $S_B = 0$ $F_A = 3C$ $F_B = C$	$S_A = C$ $S_B = 0$ $F_A = 3C$ $F_B = C$
17	$C_A = 2C_B$ $A_A > A_B$ $P_A > P_B$	$S_A = TS$ $S_B = 0$ $F_A = 2C + TS$ $F_B = 3C$	$S_A = C$ $S_B = 0$ $F_A = 3C$ $F_B = C$	$S_A = C$ $S_B = 0$ $F_A = 3C$ $F_B = C$	$S_A = A_A$ $S_B = 0$ $F_A = 2C + S_A$ $F_B = 3C$
18	$C_A = 2C_B$ $A_A > A_B$ $P_A < P_B$	$S_A = C$ $S_B = 0$ $F_A = 3C$ $F_B = C$	$S_A = C$ $S_B = 0$ $F_A = 3C$ $F_B = C$	$S_A = C$ $S_B = 0$ $F_A = 3C$ $F_B = C$	$S_A = C$ $S_B = 0$ $F_A = 3C$ $F_B = C$
where C is service time, A is arrival time, P is priority, S is start time, F is finish time, and TS is the Time Slice if RR is used.					

Table 3.3. Predicted Execution Results for Test Cases 10 - 18

Test Case	Parameters	RR	$FCFS$	SJF	$Priority$
19	$2C_A = C_B$ $A_A = A_B$ $P_A = P_B$	$S_A = 0$ or TS $S_B = TS$ or 0 $F_A = 2C - TS$ or $2C$ $F_B = 3C$ or $3C$	$S_A = 0$ or $2C$ $S_B = C$ or 0 $F_A = C$ or $3C$ $F_B = 3C$ or $2C$	$S_A = 0$ $S_B = C$ $F_A = C$ $F_B = 3C$	$S_A = 0$ or C $S_B = C$ or 0 $F_A = C$ or $3C$ $F_B = 3C$ or $2C$
20	$2C_A = C_B$ $A_A = A_B$ $P_A > P_B$	$S_A = 0$ $S_B = C$ $F_A = C$ $F_B = 3C$	$S_A = 0$ $S_B = C$ $F_A = C$ $F_B = 3C$	$S_A = 0$ $S_B = C$ $F_A = C$ $F_B = 3C$	$S_A = 0$ $S_B = C$ $F_A = C$ $F_B = 3C$
21	$2C_A = C_B$ $A_A = A_B$ $P_A < P_B$	$S_A = 2C$ $S_B = 0$ $F_A = 3C$ $F_B = 2C$	$S_A = 2C$ $S_B = 0$ $F_A = 3C$ $F_B = 2C$	$S_A = 2C$ $S_B = 0$ $F_A = 3C$ $F_B = 2C$	$S_A = 2C$ $S_B = 0$ $F_A = 3C$ $F_B = 2C$
22	$2C_A = C_B$ $A_A < A_B$ $P_A = P_B$	$S_A = 0$ $S_B = TS$ $F_A = 2C - TS$ $F_B = 3C$	$S_A = 0$ $S_B = C$ $F_A = C$ $F_B = 3C$	$S_A = 0$ $S_B = C$ $F_A = C$ $F_B = 3C$	$S_A = 0$ $S_B = C$ $F_A = C$ $F_B = 3C$
23	$2C_A = C_B$ $A_A < A_B$ $P_A > P_B$	$S_A = 0$ $S_B = C$ $F_A = C$ $F_B = 3C$	$S_A = 0$ $S_B = C$ $F_A = C$ $F_B = 3C$	$S_A = 0$ $S_B = C$ $F_A = C$ $F_B = 3C$	$S_A = 0$ $S_B = C$ $F_A = C$ $F_B = 3C$
24	$2C_A = C_B$ $A_A < A_B$ $P_A < P_B$	$S_A = 0$ $S_B = TS$ $F_A = 3C$ $F_B = 2C + TS$	$S_A = 0$ $S_B = C$ $F_A = C$ $F_B = 3C$	$S_A = 0$ $S_B = C$ $F_A = C$ $F_B = 3C$	$S_A = 0$ $S_B = A_B$ $F_A = 3C$ $F_B = 2C + S_B$
25	$2C_A = C_B$ $A_A > A_B$ $P_A = P_B$	$S_A = TS$ $S_B = 0$ $F_A = 2C$ $F_B = 3C$	$S_A = 2C$ $S_B = 0$ $F_A = 3C$ $F_B = 2C$	$S_A = 2C$ $S_B = 0$ $F_A = 3C$ $F_B = 2C$	$S_A = 2C$ $S_B = 0$ $F_A = 3C$ $F_B = 2C$
26	$2C_A = C_B$ $A_A > A_B$ $P_A > P_B$	$S_A = TS$ $S_B = 0$ $F_A = C + TS$ $F_B = 3C$	$S_A = 2C$ $S_B = 0$ $F_A = 3C$ $F_B = 2C$	$S_A = 2C$ $S_B = 0$ $F_A = 3C$ $F_B = 2C$	$S_A = A_A$ $S_B = 0$ $F_A = C + S_A$ $F_B = 3C$
27	$2C_A = C_B$ $A_A > A_B$ $P_A < P_B$	$S_A = 2C$ $S_B = 0$ $F_A = 3C$ $F_B = 2C$	$S_A = 2C$ $S_B = 0$ $F_A = 3C$ $F_B = 2C$	$S_A = 2C$ $S_B = 0$ $F_A = 3C$ $F_B = 2C$	$S_A = 2C$ $S_B = 0$ $F_A = 3C$ $F_B = 2C$

where C is service time, A is arrival time, P is priority,
S is start time, F is finish time, and TS is the Time Slice if RR is used,

Table 3.4. Predicted Execution Results for Test Cases 19 - 27

Combining a flow-time analysis for each test case to the Gantt chart observations improved the ability to distinguish between the four algorithms, but the distinction was still not as clear as desired. Still I felt that with twenty-seven test cases, it was possible to identify test case combinations which would differentiate between the algorithms. So, I summarized the expected results by grouping the task parameters together, and ordering the results by increasing start times for task A. Through this summary, some distinction between the scheduling algorithms was revealed. The summary which illustrates this distinction is provided in Table 3.5.

Parameters				Test Cases that Reveal the Algorithms			
S_A	S_B	F_A	F_B	RR	FCFS	SJF	Priority
0	TS	$2C - TS$	$2C$	1,4			
			$3C$	19,22			
		$3C$	$2C$	10,13			
			$C + TS$	15			
			$2C + TS$	24			
		$2C$	$C + TS$	6			
	A_B	$2C$	$C + S_B$				6
		$3C$	$C + S_B$				15
			$2C + S_B$				24
	C	C	$2C$	2,5	1,2,4,5,6	1,2,4,5,6	1,2,4,5
			$3C$	20,23	19,20,22,23,24	19,20,22,23,24	19,20,22,23
	$2C$	$2C$	$3C$	11,14	11,10,13,14,15	11,13,14,15	11,10,13,14
TS	0	$C + TS$	$2C$	8			
			$3C$	26			
		$2C$	$2C - TS$	1,7			
			$3C$	19,25			
		$2C + TS$	$3C$	17			
		$3C$	$2C - TS$	10,16			
A_A	0	$C + S_A$	$2C$				8
			$3C$				26
		$2C + S_A$	$3C$				17
C	0	$2C$	C	3,9	1,3,7,8,9	1,3,7,8,9	1,3,7,9
		$3C$	C	12,18	10,12,16,17,18	10,12,16,17,18	11,10,13,14
$2C$	0	$3C$	$2C$	21,27	19,21,25,26,27	21,25,26,27	19,21,25,27
NOTE: the numbers in the algorithm columns represent individual test cases.							

Table 3.5. Scheduling Algorithm Detection Summary

The table shows that there are several groupings of test case results which can be used to distinguish between the algorithms. The test cases listed at the intersection of each row and column, in the bottom right of the table, represent those test cases which will produce the set of expected results shown to the left when executed by the algorithm identified for that column. There are two possibilities of expected results for test cases 1, 10, and 19 under the RR, FCFS, and Priority columns because of equal arrival times for the tasks. Depending on whether task A or B is selected for execution first, the expected results would fall into either the top portion of the table (when task A executes first) or the bottom (when task B executes first). This is also the reason that test case 1 occurs twice under the SJF column. In spite of this duplication, there is enough distinction between the expected results to distinguish between which algorithm is used. After running all test cases under an unknown scheduling algorithm, if the algorithm is one of the four shown in the table, the actual results should match closely to the results in one of the columns. Test cases 1, 4. 6, 7, 8, 10, 13, 15, 16, 19, 22, and 24 can distinguish RR from the other algorithms. Test cases 6, 8, 15, 17, and 24 can distinguish Priority from the other algorithms. However, a problem arises when trying to distinguish between FCFS and SJF. Depending on whether task A or B is executed first in test cases 10 and 19, FCFS and SJF may produce identical results. If an algorithm executes task A first for test case 10 and task B first for test case 19, then the results would distinguish FCFS from SJF. Otherwise, there is no distinction. Therefore, an additional test case is needed to explicitly distinguish between the FCFS and SJF algorithms.

As noted earlier, I realized that these test cases could only detect the static SJF algorithm, not the dynamic one. Since Ada tasks can be dynamically created, it is not very feasible that an Ada run-time system would have the prior knowledge of task service times before execution. Though it is still unlikely that an Ada run-time system would keep track of individual task CPU burst times to use for task scheduling, I decided to add a test case to detect dynamic SJF so both forms of SJF were included.

This final test case requires six tasks: two with very short CPU burst times, two with medium burst times, and two with long CPU burst times. All tasks will be assigned equal priorities and assumed to have equal arrival times. The task parameter relationships for this test case are provided

Test Case 28			
Task ID	*CPU Burst Required*	*Arrival Time*	*Priority*
A	$\beta_A = C$	0	P
B	$\beta_B = C/2$	0	P
C	$\beta_C = C/100$	0	P
D	$\beta_D = C$	0	P
E	$\beta_E = C/2$	0	P
F	$\beta_F = C/100$	0	P
where C is some large CPU burst requirement, and β_i is CPU burst requirement for Task i			

Table 3.6. Task Parameter Relationships for Test Case 28

in Table 3.6.

The Gantt chart, as well as subsequent start and finish times, for this test case depend on the order in which the tasks are started. The initial order is not important for algorithm detection, but the relationships between start and finish times will distinguish between RR, FCFS, and dynamic SJF. Priority will not be considered in this test case because it is already addressed in the earlier test cases and no additional information would be gained by having unequal task priorities in this test case. Given that the CPU burst request occurs at least three times during task execution, and the initial order of execution is F, A, E, D, C, and B; Tables 3.7 through 3.9 show the Gantt charts for FCFS, RR, and SJF respectively.

FCFS Gantt Chart for Test Case 28						
Time	0-1	2-201	202-301	302-501	502-503	504-603
Task	F	A	E	D	C	B
Time	604-605	606-805	806-905	906-1105	1106-1107	1108-1207
Task	F	A	E	D	C	B
Time	1208-1209	1210-1409	1410-1509	1510-1709	1710-1809	1810-1811
Task	F	A	E	D	C	B
where Time is expressed in TS units, and $2TS = C/100 or TS = C/200$.						

Table 3.7. Predicted FCFS Gantt Chart for Test Case 28

The FCFS algorithm will service the jobs in the order they are queued in the ready queue (e.g. F, A, E, D, C, B) and the jobs will complete in that same order. The start and finish times

RR Gantt Chart for Test Case 28												
Time	0	1	2	3	4	5	6	7	8	9	10	11
Task	F	A	E	D	C	B	F	A	E	D	C	B
Time	12	13	14	15	16	17	18	19	20	21	22	23
Task	F	A	E	D	C	B	F	A	E	D	C	B
Time	24	25	26	27	28	29	30	31	32	33	34	35
Task	F	A	E	D	C	B	F	A	E	D	C	B
Time	36	37	38	39	. . .				1212	1213	1214	1215
Task	A	E	D	C	. . .				A	E	D	C
Time	1216	1217	. . .								1810	1811
Task	F	A	. . .								F	A
where Time is expressed in TS units, and $2TS = C/100 or TS = C/200$.												

Table 3.8. Predicted RR Gantt Chart for Test Case 28

Dynamic SJF Gantt Chart for Test Case 28						
Time	0-1	2-201	202-301	302-501	502-503	504-603
Task	F	A	E	D	C	B
Time	604-605	606-607	608-707	708-807	808-1007	1008-1207
Task	F	C	E	B	A	D
Time	1208-1209	1210-1211	1212-1311	1312-1411	1412-1611	1612-1811
Task	F	C	E	B	A	D
where Time is expressed in TS units, and $2TS = C/100 or TS = C/200$.						

Table 3.9. Predicted SJF Gantt Chart for Test Case 28

for FCFS will be staggered by the amount of CPU burst time required by each preceding task. The RR algorithm will service the jobs in the order they are queued, but it will only permit a small slice of the CPU each time through the queue. Thus, the start times of the RR algorithm will have each task staggered by a time slice. The finish times of the tasks under RR will be grouped by burst requirements, with those tasks requiring the smaller bursts finishing first. The tasks in each group will be staggered by the number of time slices needed to finish their last burst requirement (i.e.(CPU burst time / time slice) * number of remaining tasks). Finally, the Dynamic SJF algorithm will have start times which are equivalent to FCFS (it has to run the tasks through their first CPU burst). The finish times under Dynamic SJF will be grouped similar to RR, but with the tasks within each group staggered by the preceding task's burst requirement. These expected start and finish times are summarized in Tables 3.10 through 3.12.

After closely looking at these predicted results, the single test case described above should produce a distinct set of results when run by different scheduling algorithms. Thus, this test case may be the only one required to distinguish between the four scheduling algorithms of interest. I was still curious as to whether the initial twenty-seven test cases could be used to detect the RR, FCFS, and Priority algorithms. I was also curious as to how well I could model the test cases with Ada programs. In particular, I wanted to see whether I could model the equal arrival times for two Ada tasks. Finally, I wanted to see how closely the actual execution results would compare with those predicted using the Gantt charts and flow-time analysis. So, I decided to use all twenty-eight test cases, model them using Ada programs, compile the programs, and compare the actual test case results with those which are predicted here.

Predicted Test Case 28 Start and Finish Times FCFS Algorithm	
Parameter	*Predicted Result*
S_F	0
S_A	$C/100$
S_E	$C + C/100$
S_D	$C + C/100 + C/2$
S_C	$2C + C/100 + C/2$
S_B	$2C + C/50 + C/2$
F_F	$(3(L-1)/50)C + C/100$
F_A	$(3(L-1)/50)C + C + C/100$
F_E	$(3(L-1)/50)C + C + C/2 + C/100$
F_D	$(3(L-1)/50)C + 2C + C/2 + C/100$
F_C	$(3(L-1)/50)C + 2C + C/2 + C/50$
F_B	$(3(L-1)/50)C + 3C + C/50$
where L is the number of bursts required	

Table 3.10. Test Case 28 FCFS Prediction Summary

Predicted Test Case 28 Start and Finish Times RR Algorithm	
Parameter	*Predicted Result*
S_F	0
S_A	TS
S_E	$2TS$
S_D	$3TS$
S_C	$4TS$
S_B	$5TS$
F_F	$30TS + 1TS$
F_A	$36TS + (\beta_A/TS) * n)$
F_E	$36TS + (\beta_E/TS) * n)$
F_D	$36TS + (\beta_D/TS) * n)$
F_C	$30TS + 5TS$
F_B	$36TS + (\beta_B/TS) * n)$
where TS is assumed equal to $C/200$ and n is the number of remaining tasks	

Table 3.11. Test Case 28 RR Prediction Summary

Predicted Test Case 28 Start and Finish Times Dynamic SJF Algorithm	
Parameter	*Predicted Result*
S_F	0
S_A	$C/100$
S_E	$C + C/100$
S_D	$C + C/100 + C/2$
S_C	$2C + C/100 + C/2$
S_B	$2C + C/50 + C/2$
F_F	$(3(L-1)/50)C + C/100$
F_A	$(3(L-1)/50)C + C/50$
F_E	$(3(L-1)/50)C + C/50 + C/2$
F_D	$(3(L-1)/50)C + C/50 + C$
F_C	$(3(L-1)/50)C + C/50 + 2C$
F_B	$(3(L-1)/50)C + C/50 + 3C$
where L is the number of bursts required	

Table 3.12. Test Case 28 Dynamic SJF Prediction Summary

IV. Design and Development of Ada Task Schedule Detection Test Cases

The twenty-eight test cases identified for detecting task scheduling algorithms were described in Chapter III. A different combination of scheduling parameters was identified for each test case. Some of the parameters need to be controlled prior to execution, while other parameters need to be measured after execution. This chapter identifies the specific constructs used in the Ada programs to control and measure the required scheduling parameters. The Ada programs that implement the test cases are provided in Volume 2 of this thesis.

4.1 Ada Constructs Used for Implementation

The parameters of a given task, i, requiring control prior to execution are the arrival time (A_i), the service time (C_i), and the priority (P_i). The parameters which will need to be measured during task execution are the start (S_i) and finish times (F_i). The completion time (T_i) parameter will also be used, but can be derived from the start and finish times (i.e. $T_i = F_i - S_i$).

My approach for test cases 1 - 27 was to have two tasks available for uninterrupted execution. Each task is distinguished by its name (i.e. task A or B) and its associated controlled parameters (i.e. A_A, A_B, C_A, C_B, P_A, and P_B). The measured parameters (i.e. S_A, S_B, F_A, and F_B) are recorded during execution, and derived parameters (i.e. T_A and T_B) are calculated after execution. Finally, the measured and derived parameters are analyzed manually to see if they reveal the scheduling algorithm used during execution.

For test case 28, I used six tasks instead of two, I did not control task arrival times, and I did not use uninterrupted execution. In test case 28, I wanted to see how the run-time system scheduled tasks which contained fixed CPU bursts (i.e. β_A, β_B, β_C, β_D, β_E, and β_F) between run-time system interrupts (delays). In this case, the relationship between task arrivals was not as important as the relationships between task starting times and finish times.

The following discussion addresses the control of each task's arrival time, service time, CPU burst requirement, and priority. The section closes with a discussion on how an Ada task's start and finish times are measured. In each of the following sections, the Ada constructs used for test cases 1 - 27 will be discussed first and then, if different, test case 28 will be discussed.

4.1.1 Task Arrival Times. The task arrival time proved to be the most difficult parameter to control. I tried several approaches to controlling task arrival times, but none of the approaches could generate equal arrival times precisely. The three approaches considered for controlling arrival times were (1) using a task type specification, (2) using individual single task specifications & bodies, and (3) using a 'busy wait' spin test. These approaches are discussed in the following paragraphs.

Initially, I thought that it was possible to control task arrival times. The LRM states that:

> A task body defines the execution of any task that is designated by a task object of the corresponding task type. The initial part of this execution is called the *activation* of the task object, and also that of the designated task; it consists of the elaboration of the declarative part, if any, of the task body. The execution of different tasks, in particular their activation, proceeds in parallel. (12:Sec 9,5)

With regard to task activation, the LRM states that "if an object declaration that declares a task object occurs immediately within a declarative part, then the activation of the task object starts after the elaboration of the declarative part (that is, after passing the reserved word **begin** following the declarative part" (12:Sec 9,5). This implies that equal arrival times should be possible by defining a task type, and declaring several task objects of that type within the declarative portion of the parent program or procedure. Then, when the procedure's **begin** statement is reached, the task objects should be activated in parallel (i.e. they would have equal arrival times). Additionally, since activation starts after any initialization for the object created by an allocator, unequal arrival times (i.e. $A_A < A_B$) could be achieved by using two separate allocations with a delay between them. But, there are two problems associated with this approach.

The first problem is that two things cannot be done at the same time on a single-processor system. Therefore, tasks cannot be activated simultaneously as the LRM indicates, the run-time system can only activate them sequentially. The second problem is associated with the measurement of the S_i and F_i parameters. When several task objects are derived from the same task type specification, unique S_i and F_i parameters for each task object cannot be maintained. If start and finish time variables are declared in the task type specification, each task could record its own S_i and F_i, but these parameters would be lost at the termination of the tasks. Any I/O to permanently record S_i and F_i during task execution would interrupt the task, interfering with the detection of the scheduling algorithm. If start and finish time variables are declared in the parent program of the tasks, only the S_i and F_i parameters of one task would be recorded (one tasks would overwrite the parameters of the other task). This makes it impossible to record unique S_i and F_i parameters for more than one task.

Another possible approach to controlling task arrival times is to define specifications and bodies for each task. With this approach, there is no problem with S_i and F_i parameter measurements. A pair of start and finish time variables for each task can be declared in the parent procedure. Since declarations made in the parent procedure are visible to the tasks also declared in that procedure, each task can record its own S_i and F_i parameters. By creating each task object separately using allocators, this approach can also handle the unequal arrival time scenario. However, this approach has the same problem with equal arrival times as the other approach. Even though task types are not used, the run-time system still cannot activate the tasks simultaneously.

The last approach is to use a 'busy wait', and allow the parent procedure to 'start' task execution by setting a flag. With this approach, tasks are defined and declared using individual task specifications and bodies. Then, immediately upon activation, each task goes into a 'busy wait' loop. During the 'busy wait' loop, each task checks a flag and executes a delay statement if the flag hasn't been set. A state diagram, which shows the possible states of an Ada task, helps to see what really happens when the delay is used. In his book, Software Engineering with Ada, Booch describes the six possible states of an Ada task and provides a state diagram. This diagram is shown in Figure 4.1.1 (3:282).

In reference to the state diagram, initial task activation moves the task out of the *elaborated* state and into the *running* state. The delay causes a task to be blocked, and the next task in the ready queue begins execution. This implies that the 'busy wait' loop permits two tasks to swap first place position in the ready queue whenever the delay duration expires. I used **duration'small** for the delay. This value is machine dependent, but it is so small that it should not produce measurably different arrival times. When the parent program sets the flag, the task at the front of the ready queue will drop out of the 'busy wait' and begin execution. The other task will move to the front of the ready queue as soon as its delay is completed. Since either task could be at the front of the ready queue when the parent program sets the flag, each task has a equal opportunity of starting. This was as close to equal arrival times that I could come up with. This may produce unexpected results for those test cases which have unequal priorities or service times, and are expecting equal arrival times. For example, the expected results for a Priority algorithm would be that the task with the higher priority is start first; but this may not occur if the task with the higher priority is not at the front of the ready queue. But, then again, the 'right' task may be at the front of the ready queue and the actual results would correctly match those which were predicted. Analysis of actual execution results will be needed to evaluate whether this approach adequately models equal arrival times.

As noted earlier, I did not have to control arrival times for test case 28. I still used the 'busy wait' loop to insure that all tasks had an equal opportunity of being selected for execution. But in this case, instead of two tasks exchanging first place position in the ready queue, there were six tasks. Of the six tasks, two required short CPU bursts, two required medium CPU bursts, and two required long CPU bursts. Thus, after the flag was set, there was always at least one task of each

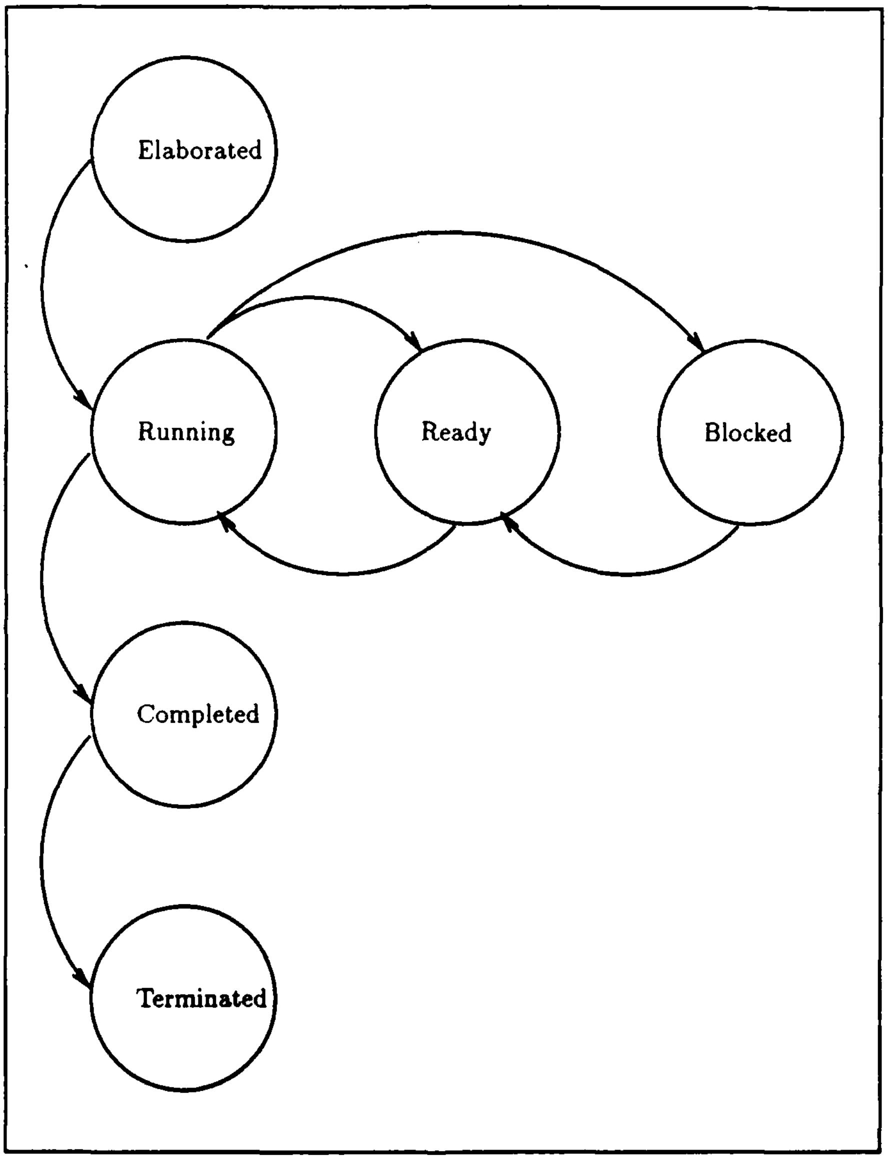

Figure 4.1. Ada Task States (3:282)

burst type in the ready queue awaiting execution.

4.1.2 Task Service Time. The task service time corresponds to the the total time which the task spends in the *running* state. The computation representing this service time was the calculation of the area of a circle, where the radius of the circle was the iteration index of the loop. The length of the service time was controlled by the number of iterations. Prior to running the test cases, C_A and C_B could be measured by running a program containing a single task. With only a single task, the run-time system executes the task until it has finished all loop iterations. By measuring S_i immediately before entering the loop and measuring F_i immediately after exiting the loop, C_i is computed by subtracting F_i from S_i. The relationship $C_A = C_B$ is achieved by having the same size loop in each task. The relationship $C_A = 2C_B$ is achieved by having task A's loop iterate twice as many times as B's loop. And $2C_A = C_B$ is achieved by having task A's loop iterate half as many times as task B's loop.

I encountered two small problems with this approach. Initially, I used integer values (ranging from 1 to 700,000) for my loop iterations, which worked fine on the mini-computer based Ada compilers. But, on the PC-based compilers, the programs would not execute because of the machine dependent constraints imposed on integer object ranges. I declared a 'SERVICE_TIME' type with the appropriate range to alleviate this first problem. The next problem was more of an inconvenience than a problem. I found that executing 700,000 iterations produced an acceptable service time on the mini computer systems, but produced too long of a service time on a PC. On the other hand, executing only 100,000 iterations on the mini computer systems did not produce a measurable service time, but produced a acceptable service time on a PC. Therefore, I had to use two separate values as my upper limit for the range of the 'SERVICE_TIME' type depending on the computer I ran the test cases on.

4.1.3 CPU Burst Requirements. In order to model CPU burst requirements, I needed to perform a continuous interval of CPU processing, then generate a task interrupt request where the task would be blocked and another task could be permitted to execute. As noted above, a continuous interval of processing can be modeled using a loop containing some 'busy' computation. For the interrupt, I used a delay statement. According to the LRM, "the execution of a delay statement evaluates the simple expression, and suspends further execution of the task that executes the delay statement, for at least the duration specified by the resulting value" (12:Sec 9,10). Placing a delay statement immediately after the busy loop, and using **Duration'small** for the simple expression, adequately modeled the CPU burst requirements. Placing this 'busy' loop within another loop of just a few iterations produces a task with several CPU bursts. In order to detect whether a Dynamic SJF algorithm is being used for task scheduling, at least two CPU bursts are needed.

The duration of the burst corresponds to the time required for the CPU to process the 'busy' loop. For test case 28, I defined a 'BURST_SIZE' type equivalent to the 'SERVICE_TIME' type for the loop iteration range. Then, I declared the loop iteration limit for the large, medium, and small bursts. A 'large burst' loop iterates to the largest number in the range and produces a CPU burst requirement which is equivalent to a single task's service time, measured above as C_i. A 'medium burst' loop iterates one half the number of times of the 'large burst' loop producing a CPU burst requirement only one half the size of the large CPU burst. Finally, a 'small burst' loop iterates only $1/100th$ the number of times of the large burst loop producing a CPU burst requirement of only $1/100th$ the size of the large CPU burst.

4.1.4 Task Priorities. The easiest scheduling parameter to implement with Ada was the priority parameter. The Ada LRM states that " a priority is associated with a task if a pragma PRIORITY (*static expression*) statement appears in the corresponding task specification" (12:Sec 9,16). The only precaution I had to take was to insure that the parent procedure had a higher priority than any of its tasks so it could set the flag needed to control arrival times. Thus, I assigned the parent procedure the highest priority. When the relationship $P_A = P_B$ was required, I assigned both tasks a priority of one less than the parent task. When the relationship $P_A < P_B$

was required, I assigned task A a priority which was two less than the parent task and B one less than the parent task. The relationship $P_A > P_B$ is clearly the opposite of the assignments noted above.

I did not use any task priorities for test case 28, but I did assign the parent task the highest possible priority. As discussed earlier, test cases 6, 8, 15, 17, 24, and 26 were developed to distinguish a Priority algorithm. Therefore, it was not necessary to include additional permutations of test case 28 with different combinations of task priorities.

4.1.5 Measurement of Start and Finish Times. In the predefined CALENDAR package, DAY_DURATION is defined as a subtype of the predefined fixed point type DURATION. Additionally, CLOCK is defined as a function which returns the current value of TIME whenever it is called, and SECONDS is defined as a function that accepts the current TIME and returns DAY_DURATION. Thus, the start time and finish time of a task (measured in seconds) can be obtained from the run-time system using a combination of these function calls. Assignment of the time to either S_i or F_i is made by invoking the function CALENDAR.SECONDS which, in turn, invokes the function CALENDAR.CLOCK. These CALENDAR functions are used in all 28 test cases to record the start and finish times of the tasks.

4.2 Overall Parent Program Structure

For each test case I used a 'main' procedure distinguished by a name which identified which test case was being modeled. Within the 'main' procedure of each test case there is a DETECT procedure which contains the required tasks. The 'main' procedure program body simply invokes the DETECT procedure, then outputs the start and finish time parameters (i.e. S_A, S_B, F_A, and F_B) after the DETECT procedure is completed. The DETECT procedure is used to prevent the main procedure from interfering with task execution. After invoking the 'DETECT' procedure, the 'main' procedure is blocked until the 'DETECT' procedure has finished. Thus, the 'main' procedure

(1) starts the DETECT procedure, (2) is blocked during task execution and does not interfere with the run-time systems ability to schedule tasks, and (3) records the measured parameters prior to program completion.

The 'DETECT' procedure contains the declarations of parameter object types, flags, and task specifications and bodies. Most of the control, and all recording of task parameters, occurs within the task bodies. The only function the DETECT procedure performs is setting the flag(s) to control arrival times of the tasks. After setting the flag(s), the DETECT procedure is blocked until its tasks complete.

Within the task bodies, the four functions described above are sequentially performed. First, the task waits at the 'busy wait' loop for the flag to be set. After the flag is set, the task records its start time using the CALENDAR functions, SECONDS and CLOCK. Next, either the 'service time' loop or the 'CPU burst' loop is executed depending on whether the task is modeling one of test cases 1 - 27, or test case 28. Finally, the finish time is recorded.

V. *Execution Results for Ada Task Scheduling Detection*

This chapter presents the results of compiling the Ada test case programs under several Ada compilers, then executing the programs. Each compiler has its own run-time system associated with it. The goal is to detect the task scheduling algorithm which is used by each Ada run-time system and validate the test cases. The compilers used were:

- the Alsys PC AT Ada Compiler, Version 3.2;
- the VAX Ada Compiler, Version 1.0;
- the Meridian AdaVantage Compiler, Version 2.1;
- the Elxsi/Verdix Ada Compiler, Version 5.4; and
- the Encore/Verdix Concurrent Ada Compiler, Version 5.5

Based on their reference manuals, I knew the scheduling algorithms used in the first three compilers and could use these to validate my test cases. I ran the test case programs on the other two compilers to further experiment with Ada task scheduling algorithm detection. Although the analysis of the execution results for a single test case does not provide conclusive evidence of which scheduling algorithm was used by the Ada run-time system, the corporate analysis of all test case results does reveal the characteristics of the algorithm used by the Ada run-time system. The analyses of the results for each Ada compiler are provided in the following sections.

5.1 Alsys PC AT Ada Compiler

I used the Alsys PC AT (v3.2) Compiler because it provided the capability of selecting a task scheduling scheme without having to recompile. According to the *Alsys PC AT Ada Compiler (v3.2) User's Guide*, setting the 'SLICE' parameter to a value greater than zero permits control of the frequency in which the task scheduler is invoked. Thus, setting the Bind/Run-time option

'SLICE' to (50ms) causes the run-time system to use a RR algorithm with a time slice of 50ms. When the Bind/Run-time 'SLICE' option is set to zero or a negative number, the task scheduler is invoked only at explicit synchronization points (2:48). Thus, a FCFS algorithm is used by the run-time system when the 'SLICE' option is set to zero. The analysis of the results obtained should determine whether the test cases reflect the chosen scheduling method. First, the results of running the test cases with a zero 'SLICE' setting (i.e. FCFS algorithm) are discussed, then the results of running with a 50ms 'SLICE' setting (i.e. RR algorithm) are discussed.

5.1.1 Results with SLICE Option Set to Zero. The results of compiling test cases 1 through 27 using the Alsys PC AT Ada Compiler (v3.2) and executing these compiled programs on a Zenith Z-248 computer system using a 0ms 'SLICE' setting are provided in Tables B.1 through B.3 of Appendix B. The following analysis is provided for these results.

Test Cases 1 & 10 results do not reveal a clear distinction between FCFS, SJF, or Priority.

Test Case 2, 11, & 20 results do not match any predicted results for the test cases. Task B starts first, then task B is preempted when task A finishes its busy wait delay. Then, task A runs to completion before task B is allowed to finish. This could occur with these test cases because of the inability to accurately model equal arrival times in Ada. Since task B is preempted to allow execution of task A, it appears as though a Priority algorithm is being used.

Test Case 3, 12, & 21 results produced the same problem discussed above for test cases 2, 11, & 20; but with opposite task execution order. It appears as though a Priority algorithm is being used.

Test Case 4, 13, 22 results do not match any predicted results for the test cases. Task A arrives first and starts, but is preempted by Task B and blocked until task B is finished. This could occur if completion of a 'DELAY' causes the currently running task to be interrupted and swapped out. Actual algorithm could be Preemptive FCFS.

Test Case 5, 9, 14, 18, 23, & 27 results do not differentiate between any of the algorithms.

Test Case 6, 8, 15, 17, 24, & 26 results do not distinguish between RR or Priority algorithms. This could be caused by the arrival time modeling used.

Test Case 7, 16, & 25 results produced the same problem discussed above for test cases 4, 13, 22; but with opposite task execution order. Actual algorithm could be Preemptive FCFS.

Test Case 19 results do not distinguish betweer FCFS or Priority algorithms.

If the completion of a delay is one of the explicit synchronization points where the task scheduler is invoked, then the results seem to indicate that a Preemptive FCFS algorithm is used. This treatment of the delay statement inhibits the capability to model the desired task arrival times using the delay and emphasizes the need for an additional test case. Since there is no conclusive evidence that any of the designated scheduling algorithms is being used, these test cases cannot be used reliably on this compiler.

Since the analysis of the results of test cases 1 - 27 did not clearly reveal which algorithm was used, I ran test case 28 to see if it could detect the FCFS algorithm. The results of compiling test case 28 using the Alsys PC AT Ada (v3.2) Compiler and executing it on the Zenith Z-248 computer system after setting the 'SLICE' option to 0 are provided in Tables B.4 through B.6 of Appendix B. I complied the test case once, but executed it three times to produce three sets of results.

The start times for these results are clearly separated by the corresponding time it would take for the preceding task to complete a CPU burst. This would imply that either SJF or FCFS is used. But, since the completion order is the same as the starting order and the separation of finish times has the same relationship as the starting times, the FCFS algorithm is revealed. Therefore, the execution sequence and the relationships of the start and finish times confirm that the Alsys PC AT Ada (v3.2) Compiler uses a FCFS algorithm to schedule tasks when the 'SLICE' parameter is set to zero. This also validates that this test case can be used for FCFS algorithm detection.

5.1.2 Results with SLICE Option Set to 50 ms. The results of compiling test cases 1 through 27 using the Alsys PC AT Ada Compiler (v3.2) and executing these compiled programs on a Zenith Z-248 computer system using a 50*ms* 'SLICE' setting are provided in Tables B.7 through B.9 of Appendix B. Throughout these results, the actual time slice appears to be greater than the .05 seconds which was selected under the run-time option. This most likely is attributable to the required context switching. Thus, I assumed that a .11*sec* difference between task start times was due to a time slice expiration, and a .05*sec* difference was due to the busy wait delay used to model task arrival times. Based on these assumptions, the following analysis is provided for the results which are shown in Tables B.7 through B.9 of Appendix B.

Test Case 1, 7, 10, 16, 19, & 25 results reveal the RR algorithm was used.

Test Case 2, 11, & 20 results do not distinguish between RR or Priority algorithms. This is due to Task B unexpectedly starting before task A. This could be caused by task B being in the Ready queue while task A is blocked due to the busy wait delay, which is related to modeling task arrival times.

Test Case 3, 12, & 21 results indicate either RR or Priority as noted above, but with the tasks swapped around.

Test Case 5, 9, 14, 18, 23, & 27 results do not differentiate between any of the algorithms.

Test Case 6, 15, & 24 results reveal the RR algorithm. The distinction between Priority is made because the start time of task *B* is greater than that which would be encountered with a Priority algorithm.

Test Case 8, 17, & 26 results reveal the RR algorithm. The distinction between Priority is made because the start time of task *A* is greater than that which would be encountered with a Priority algorithm.

Test Case 4, 13, & 22 results reveal the RR algorithm was used. But instead of task B starting after a time slice, it seems as though task B starts immediately after the busy wait. This results in task A not finishing before task B. It could be that task A is in the Ready queue and begins running; then when task B is finished with the busy wait, it preempts task A and runs a full time slice before task A has a chance to complete a full time slice. Thus, after task B is finished, task A still has some processing to complete. It definitely can't be FCFS, SJF or Priority because the start time for task *B* is not equal to 2C or 0.

Although the results of some test cases did not reveal RR, this was expected and identified in chapter III. The majority of the test cases reveal that the RR algorithm was used. This confirms that the Alsys PC AT Ada (v3.2) Compiler does use a RR algorithm when the 'Time Slice' option is set to a number greater than zero. And this validates that these test cases can detect when an Ada compiler uses a RR algorithm for task scheduling.

Although the results of the first twenty-seven test cases revealed the RR algorithm, I ran test case 28 to see if it could also reveal the correct algorithm. The results of compiling test case 28 using the Alsys PC AT Ada (v3.2) Compiler and executing it on the Zenith Z-248 computer system after setting the 'SLICE' option to $50ms$ are provided in Tables B.10 through B.12 of Appendix B. These results represent the same program compiled once, then executed three times. Analysis of these results revealed the following information.

The start times for these results are separated by corresponding time required for a time slice and the associated context switching. If either FCFS or SJF were being used, the difference between task start times would be much larger. Additionally, the reordering of the finish time sequence from that of the start time sequence clearly rules out a FCFS algorithm. The finish times of tasks A & D, tasks B & E, and tasks C & F are separately grouped in the order of shortest to largest CPU bursts. This reordering indicates that either SJF or RR was used. But, the close proximity of the task finish times within each group clearly distinguish this as RR, and not SJF. The close proximity of the task finish times within each group represent the completion of one final time slice. If a SJF algorithm were used, the difference between task finish times within each group would correspond to the time required for a final small, medium, or large CPU burst. Thus, the relationships between the start times and finish times, as well as the execution order, clearly reveal that a RR algorithm was used. This also validates that this test case can be used for RR algorithm detection

5.2 VAX Ada Compiler

I used the VAX Ada (v1.0) Compiler because it provided a 'pragma TIME_SLICE (*static expression*)' statement which is used to alter the sequence of task scheduling. Although this pragma statement is not available on all Ada compilers, it was easily inserted to provide results on an additional compiler with a known scheduling algorithm. According to the *VAX Ada Language Reference Manual*,

> The effect of enabling round-robin scheduling with pragma TIME_SLICE is defined by the following rules:
>
> - The value applies to the scheduling of every task in the program.
> - As long as an executing task is not preempted from the processor by a task of higher priority and does not become suspended, that task will execute for at most the number of seconds (approximate elapsed time) specified by the pragma. Then, if other tasks of the same priority are eligible for execution, the executing task will stop executing, and the task that has been waiting the longest will be selected for execution (11:Sec 9,22).

When the pragma TIME_SLICE is not used (or when the static expression is set to zero), the *VAX Ada Language Reference Manual* indicates that "a task is executed either until it becomes suspended or until a task of higher priority becomes eligible for execution" and "tasks of the same priority are executed in first-in first-out order (by default)" (11:Sec 9,21). Thus, when the test cases are run with the 'pragma TIME_SLICE (0.05)' statement, the run-time system should use a RR algorithm with a .05*sec* time slice to schedule tasks for execution. And when the test cases are run without the 'pragma TIME_SLICE ()' statement, the run-time system should use a FCFS algorithm when task priorities are equal and a Priority algorithm when the priorities are different.

5.2.1 Results without the 'Pragma TIME_SLICE ()' Statement. The results of compiling test cases 1 through 27 using the VAX Ada Compiler (v1.0) and executing these compiled programs on a VAX 8600 computer system without using the 'pragma TIME_SLICE ()' statement are provided in Tables B.13 through B.15 of Appendix B. The following analysis is provided for these results.

Test Cases 1, 4, 7, 13, 16, 19, 22, & 25 results do not reveal a clear distinction between FCFS, SJF, or Priority.

Test Case 2, 11, & 20 results do not match any predicted results for the test cases. Task B starts first, then task B is preempted when task A finishes its busy wait delay. Then, task A runs to completion before task B is allowed to finish. This could occur with these test cases because of the inability to accurately model equal arrival times in Ada. Since task B is preempted to allow execution of task A, it appears as though a Priority algorithm is being used.

Test Case 3, 12, & 21 results produced the same problem discussed above for test cases 2, 11, & 20; but with opposite task execution order. It appears as though a Priority algorithm is being used.

Test Case 5, 9, 14, 18, 23, & 27 results do not differentiate between any of the algorithms.

Test Case 6, 8, 15, 17, 24, & 26 results do not distinguish between RR or Priority algorithms. This could be caused by the arrival time modeling used.

Test Case 10 results do not distinguish between FCFS or Priority algorithms.

Several of the test cases reveal that a Priority algorithm scheme is used when task priorities are different. But, the FCFS algorithm is not clearly revealed when task priorities are equal. Once again, the treatment of the delay statement inhibits the capability to accurately model the task arrival times using a delay, and emphasizes the need for an additional test case.

Since the analysis of the results for test cases 1 - 27 did not clearly reveal which algorithm was used. I ran test case 28 to see if it could detect the FCFS algorithm. The results of compiling test case 28 using the VAX Ada (v1.0) Compiler and executing it on a VAX 8600 computer system without using the 'pragma TIME_SLICE ()' statement are provided in Tables B.16 through B.18 of Appendix B. I complied the test case once, but executed it three times to produce three sets of results.

The start times for these results are clearly separated by the corresponding time it would take for the preceding task to complete a CPU burst. This would imply that either SJF or FCFS is used. But, since the completion order is the same as the starting order and the separation of finish times has the same relationship as the starting times, the FCFS algorithm is revealed. Therefore, the execution sequence and the relationships of the start and finish times confirm that the VAX Ada (v1.0) Compiler uses a FCFS algorithm to schedule tasks when the 'pragma TIME_SLICE ()' is not used. This also validates that this test case can be used for FCFS algorithm detection.

5.2.2 Results with the 'Pragma TIME_SLICE (0.05)' Statement. The results of compiling test cases 1 through 27 using the VAX Ada Compiler (v1.0) and executing these compiled programs on a VAX 8600 computer system while using the 'pragma TIME_SLICE (0.05)' statement are provided in Tables B.19 through B.21 of Appendix B. The following analysis is provided for these results.

Test Case 1, 4, 7, 10, 13, 16, 19, 22, & 25 results reveal the RR algorithm was used.

Test Case 2, 11, & 20 results do not match any predicted results for the test cases. Task B starts first, then task B is preempted when task A finishes its busy wait delay. Then, task A runs to completion before task B is allowed to finish. This could occur with these test cases because of the inability to accurately model equal arrival times in Ada. Since task B is preempted to allow execution of task A, it appears as though a Priority algorithm is being used.

Test Case 3, 12, & 21 results produced the same problem discussed above for test cases 2, 11, & 20; but with opposite task execution order. It appears as though a Priority algorithm is being used.

Test Case 5, 9, 14, 18, 23, & 27 results do not differentiate between any of the algorithms.

Test Case 6, 8, 15, 17, 24, & 26 results reveal the Priority algorithm was used.

These results reveal that a RR scheduling algorithm is used. The test case results that indicate a Priority algorithm was used reveal the proper handling of tasks with unequal priorities. These results are consistent with what the manual says about task scheduling when the 'pragma TIME_SLICE (0.05)' statement is used with the VAX Ada (v1.0) Compiler.

Although the first twenty-seven test cases revealed the RR algorithm, I ran test case 28 to see if it could also reveal the correct algorithm. The results of compiling test case 28, with the pragma TIME_SLICE (0.05) statement, using the VAX Ada (v1.0) Compiler and executing it on a VAX 8600 computer system are provided in Tables B.22 through B.24 of Appendix B. I compiled the program once, then executed it three times to produce three sets of results.

These results show the same relationships as encountered with the results of the Alsys PC AT Ada Compiler with the 'SLICE' option set to 50ms. In all three runs of test case 28 the execution sequence and comparative timing of the starts and finishes reveal that the RR algorithm was used. The only problem with these results was the lack of distinction between the start times for tasks B & C, and tasks E & F. The length of execution time for the two 'Small_Burst' tasks was not long enough to be measured by the system. If given more time, I would rerun this test case with a longer burst time for the small, medium, and large burst loops. Regardless of this problem, it was still clear that the start times were separated by the corresponding time required for a time slice and context switching. The remaining discussion for finish times is the same as that provided for running test case 28 on the Alsys PC AT Ada Compiler with the 'SLICE' option set to 50ms. Overall, the execution sequence and the relationships of the start and finish times reveal that a RR algorithm is being used by the run-time system of the VAX Ada (v1.0) Compiler when the time slice pragma is used. Once again, this validates correct algorithm detection using this test case.

5.3 Meridian AdaVantage Compiler

I used the Meridian AdaVantage (v2.1) Compiler because the user manual for this compiler specified the method used for task scheduling. According to the Meridian AdaVantage (v2.1) Compiler User's Manual, this compiler's "task scheduler is not preemptive (i.e. task scheduler does not use time slicing)", instead "a single-processor round-robin prioritized scheduling system switches tasks at activations, entry calls, completions, and wait conditions" (27:61). Although the manual indicates a RR algorithm, the switching does not take place at predetermined time slice intervals. Therefore, I would be more prone to label this a FCFS algorithm where preemptions can occur when tasks request services of the run-time system. The analysis of the results obtained should determine whether this type of algorithm is actually detected. The results of compiling test cases 1 through 27 using the Meridian AdaVantage (v2.1) Compiler and executing these compiled programs on a Zenith Z-248 computer (IBM-PC/AT compatible) system are provided in Tables B.25 through B.27 of Appendix B. The following analysis is provided for these results.

Test Cases 1, 4, 7, 10, 13, 16, 22, & 25 results do not reveal a clear distinction between FCFS, SJF, or Priority algorithms.

Test Case 2, 3, 5, 9, 11, 12, 14, 18, 20, 21, 23, & 27 results do not differentiate between any of the algorithms.

Test Cases 6, 8, 15, 17, 24, & 26 results do not reveal a clear distinction between FCFS or SJF algorithms.

Test Case 19 results do not distinguish between FCFS or Priority algorithms.

There was no single test case result which revealed a unique algorithm. But, the intersection of all the test case results revealed that the Meridian AdaVantage (v2.1) Compiler most likely uses a FCFS algorithm for task scheduling. This is consistent with the discussion provided above based on the description in the user's manual.

Although the analysis of test cases 1-27 revealed a FCFS algorithm, I ran test case 28 to see if it could also detect the correct algorithm. The result of compiling test case 28 using the Meridian AdaVantage Compiler (v2.1) and executing it on the Zenith Z-248 computer system is provided in Table B.28 of Appendix B. Due to time constraints, I only ran test case 28 one time on the Meridian AdaVantage Compiler.

The start times for these results are clearly separated by the corresponding time it would take for the preceding task to complete a CPU burst. This would imply that either SJF or FCFS is used. But, since the completion order is the same as the starting order and the separation of finish times has the same relationship as the starting times, the FCFS algorithm is revealed. Therefore, the execution sequence and the relationships of the start and finish times confirm that the Meridian AdaVantage (v2.1) Compiler uses a FCFS algorithm to schedule tasks. This also validates that this test case can be used for FCFS algorithm detection.

5.4 Elxsi/Verdix Ada Compiler

I used the Elxsi/Verdix Ada Compiler because it was convenient and fast. The *Elxsi/Verdix Ada (v5.4) Development Systems Manual* points out that "by default, all Ada tasks run together as a single process (this is standard practice in Ada compilers)" (26). Although their comment does not specify a particular scheduling algorithm, it implies that a RR algorithm scheme is used to permit the tasks to 'run together'. The analysis of the results obtained should determine whether this type of algorithm is actually detected. The results of compiling test cases 1 through 27 using the Elxsi/Verdix Ada (v5.4) Compiler and executing these programs on the Elxsi computer system are provided in Tables B.29 through B.31 of Appendix B. The following analysis is provided for these results.

Test Case 1, 4 & 6 results reveal a RR algorithm. For some reason task *A* starts first, but doesn't finish first. This could be due to the inability to accurately model the task arrival times.

In any case, the algorithm cannot be FCFS, SJF or Priority because the start time for task B is not equal to 0 or C.

Test Case 2, 11, & 20 results do not distinguish between RR or Priority algorithms. This is due to task B unexpectedly starting before task A. This could be caused by task B being in the Ready queue while task A is blocked due to the busy wait delay, which is related to modeling task arrival times.

Test Case 3, 12, & 21 results indicate either RR or Priority as noted above, but with the tasks swapped around.

Test Case 5, 9, 14, 18, 23, & 27 results do not differentiate between any of the algorithms.

Test Case 7, 16, 19, & 25 results reveal RR, but the TS seems longer than it should be (could be due to multi-user aspect of computer system). It definitely can't be FCFS, SJF or Priority because the start time for task A is not equal to C.

Test Case 8, 10, 13, 15, 17, 22, 24, & 26 results reveal the RR algorithm was used.

Although there are some unexpected results due to the inability to accurately model task arrival times, it seems conclusive that a RR algorithm is being used.

Here, I ran test case 28 to validate the conclusion reached with the first twenty-seven test cases. The results of compiling test case 28 using th . Elxsi/Verdix Ada (v5.4) Compiler and executing it on the Elxsi computer system is provided in Tables B.32 through B.34 of Appendix B. I compiled the program once, then executed it three times to produce three sets of results. Analysis of these results revealed the following information.

These results showed the same relationships as were encountered with the VAX Ada (1.0) compiler. Again, the same problem with start times was encountered. And, if given more time, I would have rerun this test case with longer CPU burst times. Regardless of this problem, the start times for tasks which followed a long or medium CPU burst were separated by the time required for a time slice and context switching. If either FCFS or SJF were being used, the duration between task start times would be much larger. Additionally, the sequence of finish times, and the relationship between the finish times for tasks A & D, tasks B & E, and tasks C & F reveal a RR algorithm is being used. Thus, the execution sequence and comparative timing of the starts and finishes for this single test case support the conclusion reached with the first twenty-seven test cases.

5.5 Encore/Verdix Concurrent Ada Compiler

I used the Encore/Verdix Concurrent Ada (v5.5) Compiler to determine whether Verdix used the same scheduling algorithm in separate compilers designed for two different computer systems. Though the Encore is a parallel computer system, the Encore operating system permits the user to select the number of processors to be used. All test cases were run on the Encore computer system under a single processor environment using the Encore/Verdix Concurrent Ada (v5.5) compiler. I was not able to locate any documentation for the Encore/Verdix Concurrent Ada (v5.5) Compiler. Thus, I had no prior knowledge of which scheduling algorithm is used with this run-time system. The results of compiling test cases 1 through 27 using the Encore/Verdix Concurrent Ada (v5.5) Compiler and executing these programs on the Encore computer system are provided in Tables B.35 through B.37 of Appendix B. The following analysis is provided for these results.

Test Cases 1, 4, 7, 10, 13, 16, 22, & 25 results do not reveal a clear distinction between FCFS, SJF, or Priority algorithms.

Test Case 2, 3, 5, 9, 11, 12, 14, 18, 20, 21, 23, & 27 results do not differentiate between any of the algorithms.

Test Cases 6, 8, 15, 17, 24, & 26 results do not reveal a clear distinction between FCFS or SJF algorithms.

Test Case 19 results do not distinguish between FCFS or Priority algorithms.

There was no case where the results of an individual test case singled out a unique algorithm. But, the intersection of all the test case results indicate that a FCFS algorithm is used by the Encore/Verdix Concurrent Ada (v5.5) compiler.

After running test cases 1 - 27, I ran test case 28 to validate the conclusion noted above. The result of compiling test case 28 using the Encore/Verdix Ada (v5.5) Compiler and executing it on the Encore computer system is provided in Tables B.38 through B.40 of Appendix B. I compiled the program once, then executed it three times to produce three sets of results. Analysis of these results revealed the following information.

The start times for these results are clearly separated by the corresponding time it would take for the preceding task to complete a CPU burst. This would imply that either SJF or FCFS is used. But, since the completion order is the same as the starting order and the separation of finish times has the same relationship as the starting times, the FCFS algorithm is revealed. Therefore, the execution sequence and the comparative timing of the starts and finishes indicate that the Encore/Verdix Concurrent Ada (v5.4) Compiler uses a FCFS algorithm to schedule tasks. Thus, the results of this single test case support the conclusion reached with the first twenty-seven test cases.

5.6 Summary

The initial set of test cases (i.e. 1 thru 27) was used to successfully reveal the RR scheduling characteristics of the Alsys compiler when the 'SLICE' option was used and of the VAX Ada compiler when the 'TIME_SLICE' pragma was used. But, this initial set of test cases was only partially successful when it came to revealing FCFS characteristics. The Meridian compiler's FCFS algorithm characteristics were detected. However, this set of test cases could not be used to conclusively detect the FCFS characteristics of the Alsys compiler when the 'SLICE' option was set to zero, nor when the VAX Ada compiler was used without the TIME_SLICE pragma.

On the other hand, test case 28 was successfully used to reveal the RR algorithm characteristics of the Alsys and VAX Ada compilers when the 'SLICE' option and TIME_SLICE pragma were used, respectively. Additionally, this final test case was successfully used to reveal the FCFS algorithm characteristics of the Meridian compiler, the Alsys compiler when the 'SLICE' option was set to zero, and the VAX Ada compiler when the TIME_SLICE pragma was not used.

A summary of these findings is provided in Table 5.1.

Results Summary			
Compiler	*Algorithm Used by Compiler*	*Algorithm Revealed by Test Cases 1-27*	*Algorithm Revealed by Test Case 28*
Alsys w/out TIME SLICE	FCFS	Inconclusive	FCFS
Alsys with TIME SLICE	RR	RR	RR
VAX w/out SLICE pragma	FCFS	Inconclusive (*)	FCFS
VAX with SLICE pragma	RR	RR	RR
Meridian	FCFS	FCFS (**)	FCFS
* - Priority characteristics were revealed when $P_a \neq P_b$.			
** - after intersection of all test case results.			

Table 5.1. Execution Results Summary

VI. Conclusion and Recommendations

The goal of this thesis effort was to develop a suite of Ada programs to reveal, for any Ada compiler, the underlying task scheduling algorithm it uses. In pursuit of this goal, the following steps were completed:

- a review of the current work with Ada task scheduling;
- an examination of different approaches to scheduling algorithm detection;
- identification of the parameters needed to differentiate between five scheduling algorithms;
- design of a set of test cases to control and measure the scheduling parameters;
- development and execution of Ada programs to model the test cases; and
- analysis of the execution results to validate successful algorithm detection.

The following sections address the conclusions from this effort and recommend future research directions.

6.1 Conclusions

Based on my research of the current work with Ada task scheduling, I found that there are many problems associated with Ada task scheduling due to the ambiguity associated with the tasking rules identified in the LRM. Until these changes are made to the Ada language, the detection of the scheduling algorithm used by Ada run-time systems is very important to MCCR system designers.

My first approach for detecting an Ada compiler's task scheduling algorithm used a suite of twenty-seven different Ada programs. Each program modeled a test case in which the start and finish times of the two tasks was dependent on the relationships between their arrival times, service times, and priorities. Analysis of the results obtained with this approach disclosed that only the

RR algorithm could be distinguished. Other algorithms could not because unexpected results were encountered whenever the start and finish times of a test case were sensitive to task arrival times (precise control of task arrival times is not possible in Ada).

A second approach revealed that precise control of task arrival times was not as important for algorithm detection as originally anticipated. This approach used a single Ada program containing six tasks, and which required control of only CPU burst time. This controlled CPU burst time approach was used to correctly detect the task scheduling characteristics of algorithms used by several Ada run-time systems. The program accurately reflected that the Alsys PC AT Ada (v3.2) compiler uses a RR task scheduling algorithm when the *SLICE* option is set greater than zero, and a FCFS task scheduling algorithm when the *SLICE* run-time option is set to zero. Additionally, it reflected that the VAX Ada (v1.0) run-time system uses a RR task scheduling algorithm when the *pragma TIME_SLICE (0.05)* statement is included in the program, or a FCFS task scheduling algorithm when the *pragma TIME_SLICE (0.05)* statement is not included. Finally, the program reflected that the Meridian AdaVantage (v2.1) compiler uses a FCFS algorithm for task scheduling.

Since none of the Ada compilers used either SJF algorithm for task scheduling, the program was not validated for these. However, based on the distinct finish times which are expected when either of the SJF algorithms are used, the program should reflect SJF characteristics also.

Although I did not prove that absolute algorithm detection is possible, I've shown that it is possible to use an Ada program to distinguish one task scheduling algorithm from a restricted set of algorithms. Thus, it should be feasible to expand this program to a suite of Ada programs which will reveal, for any Ada compiler, the underlying task scheduling algorithm it uses.

6.2 Recommendations

This thesis effort has laid the groundwork for future development of an automated tool to assist DoD software designers in the development of MCCR systems using Ada. The following recommendations could improve the detection capability of the Ada program developed thus far.

I recommend adding a program to detect the characteristics of a Priority algorithm. This program would contain one additional task with a priority which is higher than the current six tasks. Additional Ada programs could be added to handle the detection of other algorithms as appropriate.

The upper bound of the 'BURST_TYPE' declaration in the DETECT procedure should be changed from a hard coded value to a parameter. With this change, the program could interactively prompt the user to enter the desired upper bound value. This value impacts the number of 'CPU burst' iterations, and subsequently impacts the length of time required to complete a given CPU burst. The size of the value should be based on whether the Ada compiler being investigated is targeted for a PC or mini-computer. This would permit execution on any size system, without having to change the hardcoded value of the upper bound and recompiling the program.

An additional future enhancement would be to automatically detect the processor speed by measuring the start and finish time of a predetermined CPU burst. Then, completion time could be automatically computed and used to determine the upper bound for the 'BURST_TYPE' declaration. During execution of the test suite, results could be recorded and then automatically analyzed to possibly predict the scheduling algorithm used. In this way, all user interaction could be removed.

The final recommendation would be to formally verify the test suite of Ada programs used to detect the task scheduling algorithms. This thesis effort did not prove that a given algorithm was used, the results produced here have only demonstrated the feasibility of this approach by revealing the algorithm characteristics exhibited by Ada run-time systems. The development of formal specifications, accompanied by a formal proof that the specifications do, in fact, distinguish between individual scheduling algorithms would improve the confidence of using this approach for scheduling algorithm detection.

6.3 Thesis Contribution

Previous attempts at dealing with the limitations associated with Ada's tasking model were aimed toward working around these limitations or changing the language. The work-arounds utilize pragmas and other inefficient constructs which can slow down program execution. The recommended changes to Ada may prove to be very slow at coming about. However, identification of an Ada compiler's task scheduling algorithm permits selection of the compiler which efficiently meets some scheduling requirements, without having to wait for Ada language changes to occur. The results of this research have demonstrated that detecting an Ada compiler's task scheduling algorithm is possible. And, it has provided a program which can be used by DoD MCCR system designers to select an Ada compiler which meets their task scheduling needs.

Appendix A. *Appendix A: Predicted Gantt Charts for Test Cases 1 through 27*

Test Case 1 ($C_A = C_B, A_A = A_B, P_A = P_B$)										
Algorithm	Expected Schedule *when* $S_A < S_B$									
RR	Time	0	1	2	3	4	5	6	7	8
	Task	A	B	A	B	A	B			
FCFS	Time	0	1	2	3	4	5	6	7	8
	Task	A	A	A	B	B	B			
SJF	Time	0	1	2	3	4	5	6	7	8
	Task	A	A	A	B	B	B			
Priority	Time	0	1	2	3	4	5	6	7	8
	Task	A	A	A	B	B	B			

Table A.1. Predicted Gantt Chart ($S_A < S_B$) for Test Case 1

Test Case 1 ($C_A = C_B, A_A = A_B, P_A = P_B$)										
Algorithm	Expected Schedule *when* $S_B < S_A$									
RR	Time	0	1	2	3	4	5	6	7	8
	Task	B	A	B	A	B	A			
FCFS	Time	0	1	2	3	4	5	6	7	8
	Task	B	B	B	A	A	A			
SJF	Time	0	1	2	3	4	5	6	7	8
	Task	B	B	B	A	A	A			
Priority	Time	0	1	2	3	4	5	6	7	8
	Task	B	B	B	A	A	A			

Table A.2. Predicted Gantt Chart ($S_B < S_A$) for Test Case 1

Test Case 2 ($C_A = C_B, A_A = A_B, P_A > P_B$)										
Algorithm	Expected Schedule									
RR	Time	0	1	2	3	4	5	6	7	8
	Task	A	A	A	B	B	B			
FCFS	Time	0	1	2	3	4	5	6	7	8
	Task	A	A	A	B	B	B			
SJF	Time	0	1	2	3	4	5	6	7	8
	Task	A	A	A	B	B	B			
Priority	Time	0	1	2	3	4	5	6	7	8
	Task	A	A	A	B	B	B			

Table A.3. Predicted Gantt Chart for Test Case 2

Test Case 3 ($C_A = C_B, A_A = A_B, P_A < P_B$)										
Algorithm	Expected Schedule									
RR	Time	0	1	2	3	4	5	6	7	8
	Task	B	B	B	A	A	A			
FCFS	Time	0	1	2	3	4	5	6	7	8
	Task	B	B	B	A	A	A			
SJF	Time	0	1	2	3	4	5	6	7	8
	Task	B	B	B	A	A	A			
Priority	Time	0	1	2	3	4	5	6	7	8
	Task	B	B	B	A	A	A			

Table A.4. Predicted Gantt Chart for Test Case 3

Test Case 4 ($C_A = C_B, A_A < A_B, P_A = P_B$)										
Algorithm	Expected Schedule									
RR	Time	0	1	2	3	4	5	6	7	8
	Task	A	B	A	B	A	B			
FCFS	Time	0	1	2	3	4	5	6	7	8
	Task	A	A	A	B	B	B			
SJF	Time	0	1	2	3	4	5	6	7	8
	Task	A	A	A	B	B	B			
Priority	Time	0	1	2	3	4	5	6	7	8
	Task	A	A	A	B	B	B			

Table A.5. Predicted Gantt Chart for Test Case 4

Test Case 5 ($C_A = C_B, A_A < A_B, P_A > P_B$)										
Algorithm	Expected Schedule									
RR	Time	0	1	2	3	4	5	6	7	8
	Task	A	A	A	B	B	B			
FCFS	Time	0	1	2	3	4	5	6	7	8
	Task	A	A	A	B	B	B			
SJF	Time	0	1	2	3	4	5	6	7	8
	Task	A	A	A	B	B	B			
Priority	Time	0	1	2	3	4	5	6	7	8
	Task	A	A	A	B	B	B			

Table A.6. Predicted Gantt Chart for Test Case 5

Test Case 6 ($C_A = C_B, A_A < A_B, P_A < P_B$)										
Algorithm	Expected Schedule									
RR	Time	0	1	2	3	4	5	6	7	8
	Task	A	B	B	B	A	A			
FCFS	Time	0	1	2	3	4	5	6	7	8
	Task	A	A	A	B	B	B			
SJF	Time	0	1	2	3	4	5	6	7	8
	Task	A	A	A	B	B	B			
Priority	Time	0	1	2	3	4	5	6	7	8
	Task	A	B	B	B	A	A			

Table A.7. Predicted Gantt Chart for Test Case 6

Test Case 7 ($C_A = C_B, A_A > A_B, P_A = P_B$)										
Algorithm	Expected Schedule									
RR	Time	0	1	2	3	4	5	6	7	8
	Task	B	A	B	A	B	A			
FCFS	Time	0	1	2	3	4	5	6	7	8
	Task	B	B	B	A	A	A			
SJF	Time	0	1	2	3	4	5	6	7	8
	Task	B	B	B	A	A	A			
Priority	Time	0	1	2	3	4	5	6	7	8
	Task	B	B	B	A	A	A			

Table A.8. Predicted Gantt Chart for Test Case 7

Test Case 8 ($C_A = C_B, A_A > A_B, P_A > P_B$)										
Algorithm	Expected Schedule									
RR	Time	0	1	2	3	4	5	6	7	8
	Task	B	A	A	A	B	B			
FCFS	Time	0	1	2	3	4	5	6	7	8
	Task	B	B	B	A	A	A			
SJF	Time	0	1	2	3	4	5	6	7	8
	Task	B	B	B	A	A	A			
Priority	Time	0	1	2	3	4	5	6	7	8
	Task	B	A	A	A	B	B			

Table A.9. Predicted Gantt Chart for Test Case 8

Test Case 9 ($C_A = C_B, A_A > A_B, P_A < P_B$)										
Algorithm	Expected Schedule									
RR	Time	0	1	2	3	4	5	6	7	8
	Task	B	B	B	A	A	A			
FCFS	Time	0	1	2	3	4	5	6	7	8
	Task	B	B	B	A	A	A			
SJF	Time	0	1	2	3	4	5	6	7	8
	Task	B	B	B	A	A	A			
Priority	Time	0	1	2	3	4	5	6	7	8
	Task	B	B	B	A	A	A			

Table A.10. Predicted Gantt Chart for Test Case 9

Test Case 10 ($C_A = 2C_B, A_A = A_B, P_A = P_B$)										
Algorithm	Expected Schedule *when* $S_A < S_B$									
RR	Time	0	1	2	3	4	5	6	7	8
	Task	A	B	A	B	A	B	A	A	A
FCFS	Time	0	1	2	3	4	5	6	7	8
	Task	A	A	A	A	A	A	B	B	B
SJF	Time	0	1	2	3	4	5	6	7	8
	Task	*not applicable*								
Priority	Time	0	1	2	3	4	5	6	7	8
	Task	A	A	A	A	A	A	B	B	B

Table A.11. Predicted Gantt Chart *($S_A < S_B$)* for Test Case 10

Test Case 10 ($C_A = 2C_B, A_A = A_B, P_A = P_B$)										
Algorithm	Expected Schedule *when* $S_C < S_A$									
RR	Time	0	1	2	3	4	5	6	7	8
	Task	B	A	B	A	B	A	A	A	A
FCFS	Time	0	1	2	3	4	5	6	7	8
	Task	B	B	B	A	A	A	A	A	A
SJF	Time	0	1	2	3	4	5	6	7	8
	Task	B	B	B	A	A	A	A	A	A
Priority	Time	0	1	2	3	4	5	6	7	8
	Task	B	B	B	A	A	A	A	A	A

Table A.12. Predicted Gantt Chart *($S_B < S_A$)* for Test Case 10

Test Case 11 ($C_A = 2C_B, A_A = A_B, P_A > P_B$)										
Algorithm	Expected Schedule									
RR	Time	0	1	2	3	4	5	6	7	8
	Task	A	A	A	A	A	A	B	B	B
FCFS	Time	0	1	2	3	4	5	6	7	8
	Task	A	A	A	A	A	A	B	B	B
SJF	Time	0	1	2	3	4	5	6	7	8
	Task	A	A	A	A	A	A	B	B	B
Priority	Time	0	1	2	3	4	5	6	7	8
	Task	A	A	A	A	A	A	B	B	B

Table A.13. Predicted Gantt Chart for Test Case 11

Test Case 12 ($C_A = 2C_B, A_A = A_B, P_A < P_B$)										
Algorithm	Expected Schedule									
RR	Time	0	1	2	3	4	5	6	7	8
	Task	B	B	B	A	A	A	A	A	A
FCFS	Time	0	1	2	3	4	5	6	7	8
	Task	B	B	B	A	A	A	A	A	A
SJF	Time	0	1	2	3	4	5	6	7	8
	Task	B	B	B	A	A	A	A	A	A
Priority	Time	0	1	2	3	4	5	6	7	8
	Task	B	B	B	A	A	A	A	A	A

Table A.14. Predicted Gantt Chart for Test Case 12

Test Case 13 ($C_A = 2C_B, A_A < A_B, P_A = P_B$)										
Algorithm	Expected Schedule									
RR	Time	0	1	2	3	4	5	6	7	8
	Task	A	B	A	B	A	B	A	A	A
FCFS	Time	0	1	2	3	4	5	6	7	8
	Task	A	A	A	A	A	A	B	B	B
SJF	Time	0	1	2	3	4	5	6	7	8
	Task	A	A	A	A	A	A	B	B	B
Priority	Time	0	1	2	3	4	5	6	7	8
	Task	A	A	A	A	A	A	B	B	B

Table A.15. Predicted Gantt Chart for Test Case 13

Test Case 14 ($C_A = 2C_B, A_A < A_B, P_A > P_B$)										
Algorithm	Expected Schedule									
RR	Time	0	1	2	3	4	5	6	7	8
	Task	A	A	A	A	A	A	B	B	B
FCFS	Time	0	1	2	3	4	5	6	7	8
	Task	A	A	A	A	A	A	B	B	B
SJF	Time	0	1	2	3	4	5	6	7	8
	Task	A	A	A	A	A	A	B	B	B
Priority	Time	0	1	2	3	4	5	6	7	8
	Task	A	A	A	A	A	A	B	B	B

Table A.16. Predicted Gantt Chart for Test Case 14

Test Case 15 ($C_A = 2C_B, A_A < A_B, P_A < P_B$)										
Algorithm	Expected Schedule									
RR	Time	0	1	2	3	4	5	6	7	8
	Task	A	B	B	B	A	A	A	A	A
FCFS	Time	0	1	2	3	4	5	6	7	8
	Task	A	A	A	A	A	A	B	B	B
SJF	Time	0	1	2	3	4	5	6	7	8
	Task	A	A	A	A	A	A	B	B	B
Priority	Time	0	1	2	3	4	5	6	7	8
	Task	A	B	B	B	A	A	A	A	A

Table A.17. Predicted Gantt Chart for Test Case 15

Test Case 16 ($C_A = 2C_B, A_A > A_B, P_A = P_B$)										
Algorithm	Expected Schedule									
RR	Time	0	1	2	3	4	5	6	7	8
	Task	B	A	B	A	B	A	A	A	A
FCFS	Time	0	1	2	3	4	5	6	7	8
	Task	B	B	B	A	A	A	A	A	A
SJF	Time	0	1	2	3	4	5	6	7	8
	Task	B	B	B	A	A	A	A	A	A
Priority	Time	0	1	2	3	4	5	6	7	8
	Task	B	B	B	A	A	A	A	A	A

Table A.18. Predicted Gantt Chart for Test Case 16

Test Case 17 ($C_A = 2C_B, A_A > A_B, P_A > P_B$)										
Algorithm	Expected Schedule									
RR	Time	0	1	2	3	4	5	6	7	8
	Task	B	A	A	A	A	A	A	B	B
FCFS	Time	0	1	2	3	4	5	6	7	8
	Task	B	B	B	A	A	A	A	A	A
SJF	Time	0	1	2	3	4	5	6	7	8
	Task	B	B	B	A	A	A	A	A	A
Priority	Time	0	1	2	3	4	5	6	7	8
	Task	B	A	A	A	A	A	A	B	B

Table A.19. Predicted Gantt Chart for Test Case 17

Test Case 18 ($C_A = 2C_B, A_A > A_B, P_A < P_B$)										
Algorithm	Expected Schedule									
RR	Time	0	1	2	3	4	5	6	7	8
	Task	B	B	B	A	A	A	A	A	A
FCFS	Time	0	1	2	3	4	5	6	7	8
	Task	B	B	B	A	A	A	A	A	A
SJF	Time	0	1	2	3	4	5	6	7	8
	Task	B	B	B	A	A	A	A	A	A
Priority	Time	0	1	2	3	4	5	6	7	8
	Task	B	B	B	A	A	A	A	A	A

Table A.20. Predicted Gantt Chart for Test Case 18

Test Case 19 ($2C_A = C_B, A_A = A_B, P_A = P_B$)										
Algorithm	Expected Schedule *when* $S_A < S_B$									
RR	Time	0	1	2	3	4	5	6	7	8
	Task	A	B	A	B	A	B	B	B	B
FCFS	Time	0	1	2	3	4	5	6	7	8
	Task	A	A	A	B	B	B	B	B	B
SJF	Time	0	1	2	3	4	5	6	7	8
	Task	A	A	A	B	B	B	B	B	B
Priority	Time	0	1	2	3	4	5	6	7	8
	Task	A	A	A	B	B	B	B	B	B

Table A.21. Predicted Gantt Chart *($S_A < S_B$)* for Test Case 19

Test Case 19 ($2C_A = C_B, A_A = A_B, P_A = P_B$)										
Algorithm	Expected Schedule *when* $S_B < S_A$									
RR	Time	0	1	2	3	4	5	6	7	8
	Task	B	A	B	A	B	A	B	B	B
FCFS	Time	0	1	2	3	4	5	6	7	8
	Task	B	B	B	B	B	B	A	A	A
SJF	Time	0	1	2	3	4	5	6	7	8
	Task	B	B	B	B	B	B	A	A	A
Priority	Time	0	1	2	3	4	5	6	7	8
	Task	B	B	B	B	B	B	A	A	A

Table A.22. Predicted Gantt Chart *($S_B < S_A$)* for Test Case 19

Test Case 20 ($2C_A = C_B, A_A = A_B, P_A > P_B$)										
Algorithm	Expected Schedule									
RR	Time	0	1	2	3	4	5	6	7	8
	Task	A	A	A	B	B	B	B	B	B
FCFS	Time	0	1	2	3	4	5	6	7	8
	Task	A	A	A	B	B	B	B	B	B
SJF	Time	0	1	2	3	4	5	6	7	8
	Task	A	A	A	B	B	B	B	B	B
Priority	Time	0	1	2	3	4	5	6	7	8
	Task	A	A	A	B	B	B	B	B	B

Table A.23. Predicted Gantt Chart for Test Case 20

Test Case 21 ($2C_A = C_B, A_A = A_B, P_A < P_B$)										
Algorithm	Expected Schedule									
RR	Time	0	1	2	3	4	5	6	7	8
	Task	B	B	B	B	B	B	A	A	A
FCFS	Time	0	1	2	3	4	5	6	7	8
	Task	B	B	B	B	B	B	A	A	A
SJF	Time	0	1	2	3	4	5	6	7	8
	Task	B	B	B	B	B	B	A	A	A
Priority	Time	0	1	2	3	4	5	6	7	8
	Task	B	B	B	B	B	B	A	A	A

Table A.24. Predicted Gantt Chart for Test Case 21

Test Case 22 ($2C_A = C_B, A_A < A_B, P_A = P_B$)										
Algorithm	Expected Schedule									
RR	Time	0	1	2	3	4	5	6	7	8
	Task	A	B	A	B	A	B	B	B	B
FCFS	Time	0	1	2	3	4	5	6	7	8
	Task	A	A	A	B	B	B	B	B	B
SJF	Time	0	1	2	3	4	5	6	7	8
	Task	A	A	A	B	B	B	B	B	B
Priority	Time	0	1	2	3	4	5	6	7	8
	Task	A	A	A	B	B	B	B	B	B

Table A.25. Predicted Gantt Chart for Test Case 22

Test Case 23 ($2C_A = C_B, A_A < A_B, P_A > P_B$)										
Algorithm	Expected Schedule									
RR	Time	0	1	2	3	4	5	6	7	8
	Task	A	A	A	B	B	B	B	B	B
FCFS	Time	0	1	2	3	4	5	6	7	8
	Task	A	A	A	B	B	B	B	B	B
SJF	Time	0	1	2	3	4	5	6	7	8
	Task	A	A	A	B	B	B	B	B	B
Priority	Time	0	1	2	3	4	5	6	7	8
	Task	A	A	A	B	B	B	B	B	B

Table A.26. Predicted Gantt Chart for Test Case 23

Test Case 24 ($2C_A = C_B, A_A < A_B, P_A < P_B$)										
Algorithm	Expected Schedule									
RR	Time	0	1	2	3	4	5	6	7	8
	Task	A	B	B	B	B	B	B	A	A
FCFS	Time	0	1	2	3	4	5	6	7	8
	Task	A	A	A	B	B	B	B	B	B
SJF	Time	0	1	2	3	4	5	6	7	8
	Task	A	A	A	B	B	B	B	B	B
Priority	Time	0	1	2	3	4	5	6	7	8
	Task	A	B	B	B	B	B	B	A	A

Table A.27. Predicted Gantt Chart for Test Case 24

Test Case 25 ($2C_A = C_B, A_A > A_B, P_A = P_B$)										
Algorithm	Expected Schedule									
RR	Time	0	1	2	3	4	5	6	7	8
	Task	B	A	B	A	B	A	B	B	B
FCFS	Time	0	1	2	3	4	5	6	7	8
	Task	B	B	B	B	B	B	A	A	A
SJF	Time	0	1	2	3	4	5	6	7	8
	Task	B	B	B	B	B	B	A	A	A
Priority	Time	0	1	2	3	4	5	6	7	8
	Task	B	B	B	B	B	B	A	A	A

Table A.28. Predicted Gantt Chart for Test Case 25

Test Case 26 ($2C_A = C_B, A_A > A_B, P_A > P_B$)										
Algorithm	Expected Schedule									
RR	Time	0	1	2	3	4	5	6	7	8
	Task	B	A	A	A	B	B	B	B	B
FCFS	Time	0	1	2	3	4	5	6	7	8
	Task	B	B	B	B	B	B	A	A	A
SJF	Time	0	1	2	3	4	5	6	7	8
	Task	B	B	B	B	B	B	A	A	A
Priority	Time	0	1	2	3	4	5	6	7	8
	Task	B	A	A	A	B	B	B	B	B

Table A.29. Predicted Gantt Chart for Test Case 26

Test Case 27 ($2C_A = C_B, A_A > A_B, P_A < P_B$)										
Algorithm	Expected Schedule									
RR	Time	0	1	2	3	4	5	6	7	8
	Task	B	B	B	B	B	B	A	A	A
FCFS	Time	0	1	2	3	4	5	6	7	8
	Task	B	B	B	B	B	B	A	A	A
SJF	Time	0	1	2	3	4	5	6	7	8
	Task	B	B	B	B	B	B	A	A	A
Priority	Time	0	1	2	3	4	5	6	7	8
	Task	B	B	B	B	B	B	A	A	A

Table A.30. Predicted Gantt Chart for Test Case 27

Appendix B. *Appendix B: Test Case Execution Results*

Actual Results of Running Test Cases 1-9 *using Alsys PC AT Ada Compiler, Version 3.2* *with a Slice Option of 0 seconds*					
Test Case		Actual Measured Results	Normalized Results	$F_i - S_i$	Execution Sequence
1	S_A	37488.7890	6.869	6.921	$S_B - F_B - S_A - F_A$
	F_A	37495.7100	13.79		
	S_B	37481.9200	0	6.869	
	F_B	37488.7890	6.869		
2	S_A	37543.9300	.05	6.87	$S_B - S_A - F_A - F_B$
	F_A	37550.8000	6.92		
	S_B	37543.8800	0	13.779	
	F_B	37557.6590	13.779		
3	S_A	37594.3500	0	13.79	$S_A - S_B - F_B - F_A$
	F_A	37608.1400	13.79		
	S_B	37594.4600	.11	6.869	
	F_B	37601.3290	6.979		
4	S_A	37612.0900	0	13.63	$S_A - S_B - F_B - F_A$
	F_A	37625.7200	13.63		
	S_B	37612.1490	.059	6.811	
	F_B	37618.9600	6.87		
5	S_A	37629.7790	0	6.811	$S_A - F_A - S_B - F_B$
	F_A	37636.5900	6.811		
	S_B	37636.5900	6.811	6.809	
	F_B	37643.3990	13.62		
6	S_A	37678.6090	0	13.62	$S_A - S_B - F_B - F_A$
	F_A	37692.2290	13.62		
	S_B	37678.7700	.161	6.809	
	F_B	37685.5790	6.97		
7	S_A	37696.2890	.109	6.82	$S_B - S_A - F_A - F_B$
	F_A	37703.1090	6.9299		
	S_B	37696.1800	0	13.63	
	F_B	37709.8100	13.63		
8	S_A	37713.9300	.17	6.809	$S_B - S_A - F_A - F_B$
	F_A	37720.7390	6.979		
	S_B	37713.7600	0	13.679	
	F_B	37727.4390	13.679		
9	S_A	37738.2600	6.811	6.809	$S_B - F_B - S_A - F_A$
	F_A	37745.0690	13.62		
	S_B	37731.4490	0	6.811	
	F_B	37738.2600	6.811		

Table B.1. Alsys PC AT Ada Compiler Results

Actual Results of Running Test Cases 10-18 *using Alsys PC AT Ada Compiler, Version 3.2 with a Slice Option of 0 seconds*					
Test Case		Actual Measured Results	Normalized Results	$F_i - S_i$	Execution Sequence
10	S_A	37993.2200	6.92	13.729	$S_B - F_B - S_A - F_A$
	F_A	38006.9490	20.649		
	S_B	37986.3000	0	6.92	
	F_B	37993.2200	6.92		
11	S_A	38011.0200	.06	13.73	$S_B - S_A - F_A - F_B$
	F_A	38024.7500	13.79		
	S_B	38010.9600	0	20.649	
	F_B	38031.6090	20.649		
12	S_A	38062.0390	0	20.601	$S_A - S_B - F_B - F_A$
	F_A	38082.6400	20.601		
	S_B	38062.1000	.061	6.86	
	F_B	38068.9600	6.921		
13	S_A	38091.5390	0	20.49	$S_A - S_B - F_B - F_A$
	F_A	38112.0290	20.49		
	S_B	38091.5900	.051	6.809	
	F_B	38098.3990	6.86		
14	S_A	38133.5000	0	13.68	$S_A - F_A - S_B - F_B$
	F_A	38147.1800	13.68		
	S_B	38147.1800	13.68	6.809	
	F_B	38153.9890	20.489		
15	S_A	38157.9390	0	20.491	$S_A - S_B - F_B - F_A$
	F_A	38178.4300	20.491		
	S_B	38158.1090	.17	6.811	
	F_B	38164.9200	6.981		
16	S_A	38182.5000	.11	13.67	$S_B - S_A - F_A - F_B$
	F_A	38196.1700	13.78		
	S_B	38182.3900	0	20.54	
	F_B	38202.9300	20.54		
17	S_A	38310.1990	.17	13.67	$S_B - S_A - F_A - F_B$
	F_A	38323.8690	13.84		
	S_B	38310.0290	0	20.54	
	F_B	38330.5690	20.54		
18	S_A	38341.2790	6.809	13.681	$S_B - F_B - S_A - F_A$
	F_A	38354.9600	20.49		
	S_B	38334.4700	0	6.809	
	F_B	38341.2790	6.809		

Table B.2. Alsys PC AT Ada Compiler Results (Cont'd)

Actual Results of Running Test Cases 19-27 *using Alsys PC AT Ada Compiler, Version 3.2 with a Slice Option of 0 seconds*					
Test Case		Actual Measured Results	Normalized Results	$F_i - S_i$	Execution Sequence
19	S_A	38549.1800	13.79	6.859	$S_B - F_B - S_A - F_A$
	F_A	38556.0390	20.649		
	S_B	38535.3900	0	13.79	
	F_B	38549.1800	13.79		
20	S_A	38560.0000	.061	6.859	$S_B - S_A - F_A - F_B$
	F_A	38566.8590	6.92		
	S_B	38559.6490	0	20.71	
	F_B	38580.6490	20.71		
21	S_A	38588.8290	0	20.71	$S_A - S_B - F_B - F_A$
	F_A	38609.5390	20.71		
	S_B	38588.8900	.061	13.729	
	F_B	38602.6190	13.79		
22	S_A	38613.0390	0	20.539	$S_A - S_B - F_B - F_A$
	F_A	38634.0390	20.539		
	S_B	38613.2290	.05	13.679	
	F_B	38627.2290	13.729		
23	S_A	38712.1990	0	6.811	$S_A - F_A - S_B - F_B$
	F_A	38719.0100	6.811		
	S_B	38719.0100	6.811	13.729	
	F_B	38732.7390	20.54		
24	S_A	38743.7790	0	20.601	$S_A - S_B - F_B - F_A$
	F_A	38764.3800	20.601		
	S_B	38743.9390	.16	13.68	
	F_B	38757.6190	13.84		
25	S_A	38768.8800	.11	6.809	$S_B - S_A - F_A - F_B$
	F_A	38775.6890	6.919		
	S_B	38768.7700	0	20.54	
	F_B	38789.3100	20.54		
26	S_A	38793.6000	.17	6.809	$S_B - S_A - F_A - F_B$
	F_A	38800.4090	6.979		
	S_B	38793.4300	0	20.599	
	F_B	38814.0290	20.599		
27	S_A	38844.0690	13.67	6.87	$S_B - F_B - S_A - F_A$
	F_A	38850.9390	20.54		
	S_B	38830.3990	0	13.67	
	F_B	38844.0690	13.67		

Table B.3. Alsys PC AT Ada Compiler Results (Cont'd)

First Run of Test Case 28
using Alsys PC AT Ada Compiler, Version 3.2
with a Slice Option of 0 seconds

Parameter	Actual Measured Results	Normalized Results	$F_i - S_i$
S_A	54053.1590	.159	180.54
F_A	54233.6990	180.699	
S_B	54087.7600	34.76	173.679
F_B	54261.4390	208.439	
S_C	54087.6490	34.649	166.871
F_C	54254.5200	201.52	
S_D	54073.8690	20.869	180.54
F_D	54254.4090	201.409	
S_E	54066.9490	13.949	173.67
F_E	54240.6190	187.619	
S_F	54053.0000	0	166.92
F_F	54219.9200	166.92	
Execution Sequence: $S_F - S_A - S_E - S_D - S_C - S_B - F_F - F_A - F_E - F_D - F_C - F_B$			

Table B.4. Alsys PC AT Ada Compiler Results (Cont'd)

Second Run of Test Case 28
using Alsys PC AT Ada Compiler, Version 3.2
with a Slice Option of 0 seconds

Parameter	Actual Measured Results	Normalized Results	$F_i - S_i$
S_A	54589.6700	.11	180.54
F_A	54470.2100	180.65	
S_B	54324.3290	34.769	173.671
F_B	54498.0000	208.44	
S_C	54324.1590	34.599	166.87
F_C	54491.0290	201.469	
S_D	54310.3800	20.82	180.54
F_D	54490.9200	201.36	
S_E	54303.4600	13.9	173.67
F_E	54477.1300	187.57	
S_F	54289.5600	0	166.86
F_F	54456.4200	166.86	
Execution Sequence: $S_F - S_A - S_E - S_D - S_C - S_B - F_F - F_A - F_E - F_D - F_C - F_B$			

Table B.5. Alsys PC AT Ada Compiler Results (Cont'd)

Third Run of Test Case 28 *using Alsys PC AT Ada Compiler, Version 3.2* *with a Slice Option of 0 seconds*			
Parameter	Actual Measured Results	Normalized Results	$F_i - S_i$
S_A	54523.2100	.11	180.54
F_A	54703.7500	180.65	
S_B	54557.8190	34.719	173.67
F_B	54731.4890	208.389	
S_C	54557.7100	34.61	166.859
F_C	54724.5690	201.469	
S_D	54543.9200	20.82	180.54
F_D	54724.4600	201.36	
S_E	54537.0000	13.9	173.67
F_E	54710.6700	187.57	
S_F	54523.1000	0	166.87
F_F	54689.9700	166.87	
Execution Sequence: $S_F - S_A - S_E - S_D - S_C - S_B$ $F_F - F_A - F_E - F_D - F_C - F_B$			

Table B.6. Alsys PC AT Ada Compiler Results (Cont'd)

Actual Results of Running Test Cases 1-9 *using Alsys PC AT Ada Compiler, Version 3.2* *with a Slice Option of 50 ms.*					
Test Case		Actual Measured Results	Normalized Results	$F_i - S_i$	Execution Sequence
1	S_A	47543.3900	.111	13.95	$S_B - S_A - F_B - F_A$
	F_A	47557.3400	14.061		
	S_B	47543.2790	0	13.95	
	F_B	47557.2290	13.95		
2	S_A	47561.3000	.061	6.859	$S_B - S_A - F_A - F_B$
	F_A	47568.1590	6.92		
	S_B	47561.2390	0	13.79	
	F_B	47575.0290	13.79		
3	S_A	47677.8990	0	13.851	$S_A - S_B - F_B - F_A$
	F_A	47691.7500	13.851		
	S_B	47677.8990	.061	6.869	
	F_B	47684.8290	6.93		
4	S_A	47701.9600	0	13.729	$S_A - S_B - F_B - F_A$
	F_A	47715.6890	13.729		
	S_B	47702.0200	.06	13.68	
	F_B	47715.6400	13.62		
5	S_A	47726.3500	0	6.809	$S_A - F_A - S_B - F_B$
	F_A	47733.1590	6.809		
	S_B	47733.1590	6.809	6.87	
	F_B	47740.0290	13.679		
6	S_A	47760.6800	0	13.84	$S_A - S_B - F_B - F_A$
	F_A	47774.5200	13.84		
	S_B	47760.8400	.16	6.809	
	F_B	47767.6490	6.969		
7	S_A	47778.5290	.109	13.62	$S_B - S_A - F_B - F_A$
	F_A	47792.0900	13.729		
	S_B	47778.4200	0	13.76	
	F_B	47792.0900	13.76		
8	S_A	47826.5900	.16	6.81	$S_B - S_A - F_A - F_B$
	F_A	47833.5600	6.97		
	S_B	47826.5900	0	13.67	
	F_B	47840.2600	13.67		
9	S_A	47850.9700	6.811	6.87	$S_B - F_B - S_A - F_A$
	F_A	47857.8400	13.681		
	S_B	47844.1590	0	6.811	
	F_B	47850.9700	6.811		

Table B.7. **Alsys PC AT Ada Compiler Results** (Cont'd)

Actual Results of Running Test Cases 10-18 *using Alsys PC AT Ada Compiler, Version 3.2 with a Slice Option of 50 ms.*					
Test Case		Actual Measured Results	Normalized Results	$F_i - S_i$	Execution Sequence
10	S_A	48112.1990	.109	20.761	$S_B - S_A - F_B - F_A$
	F_A	48132.9600	20.87		
	S_B	48112.0900	0	14.01	
	F_B	48126.1000	14.01		
11	S_A	48136.8590	.049	13.731	$S_B - S_A - F_A - F_B$
	F_A	48150.5900	13.78		
	S_B	48136.8100	0	20.65	
	F_B	48157.4600	20.65		
12	S_A	48204.8590	0	20.65	$S_A - S_B - F_B - F_A$
	F_A	48225.5100	20.65		
	S_B	48204.9090	.05	6.87	
	F_B	48211.7790	6.92		
13	S_A	48229.6300	0	20.599	$S_A - S_B - F_B - F_A$
	F_A	48250.2290	20.599		
	S_B	48229.6890	.059	13.731	
	F_B	48243.4200	13.79		
14	S_A	48261.5390	0	13.681	$S_A - F_A - S_B - F_B$
	F_A	48275.2200	13.681		
	S_B	48275.2200	13.681	6.809	
	F_B	48282.0290	20.49		
15	S_A	48338.2700	0	20.549	$S_A - S_B - F_B - F_A$
	F_A	48358.8190	20.549		
	S_B	48338.4390	.169	6.811	
	F_B	48345.2500	6.98		
16	S_A	48362.9390	.11	20.481	$S_B - S_A - F_B - F_A$
	F_A	48383.4200	20.591		
	S_B	48362.8290	0	13.62	
	F_B	48376.4490	13.62		
17	S_A	48387.4300	.16	13.679	$S_B - S_A - F_A - F_B$
	F_A	48401.1090	13.839		
	S_B	48387.2700	0	20.54	
	F_B	48407.8100	20.54		
18	S_A	48425.1700	6.811	13.67	$S_B - F_B - S_A - F_A$
	F_A	48438.8400	20.481		
	S_B	48418.3590	0	6.811	
	F_B	48425.1700	6.811		

Table B.8. Alsys PC AT Ada Compiler Results (Cont'd)

Actual Results of Running Test Cases 19-27 *using Alsys PC AT Ada Compiler, Version 3.2 with a Slice of 50 ms.*					
Test Case		Actual Measured Results	Normalized Results	$F_i - S_i$	Execution Sequence
19	S_A	48657.6090	.109	13.84	$S_B - S_A - F_A - F_B$
	F_A	48671.4490	13.949		
	S_B	48657.5000	0	20.76	
	F_B	48678.2600	20.76		
20	S_A	48682.3290	.059	6.86	$S_B - S_A - F_A - F_B$
	F_A	48689.1890	6.919		
	S_B	48682.2700	0	20.709	
	F_B	48702.9790	20.709		
21	S_A	48715.2290	0	20.701	$S_A - S_B - F_B - F_A$
	F_A	48735.9300	20.701		
	S_B	48715.2790	.05	13.79	
	F_B	48729.0690	13.84		
22	S_A	48739.8900	0	13.729	$S_A - S_B - F_A - F_B$
	F_A	48753.6190	13.729		
	S_B	48739.9390	.049	20.6	
	F_B	48760.5390	20.649		
23	S_A	48786.0290	0	6.811	$S_A - F_A - S_B - F_B$
	F_A	48792.8400	6.811		
	S_B	48792.8400	6.811	13.729	
	F_B	48806.5690	20.54		
24	S_A	48861.2700	0	20.599	$S_A - S_B - F_B - F_A$
	F_A	48881.8690	20.599		
	S_B	48861.4390	.169	13.731	
	F_B	48875.1700	13.9		
25	S_A	48885.8800	.11	13.729	$S_B - S_A - F_A - F_B$
	F_A	48899.4200	13.839		
	S_B	48885.7700	0	20.65	
	F_B	48906.4200	20.65		
26	S_A	48921.0900	.17	6.809	$S_B - S_A - F_A - F_B$
	F_A	48927.8990	6.979		
	S_B	48920.9200	0	20.6	
	F_B	48941.5200	20.6		
27	S_A	48959.2100	13.731	6.81	$S_B - F_B - S_A - F_A$
	F_A	48966.0200	20.541		
	S_B	48945.4790	0	13.731	
	F_B	48959.2100	13.731		

Table B.9. Alsys PC AT Ada Compiler Results (Cont'd)

First Run of Test Case 28 *using Alsys PC AT Ada Compiler, Version 3.2 with a Slice Option of 50 ms.*			
Parameter	Actual Measured Results	Normalized Results	$F_i - S_i$
S_A	54826.3500	.111	209.92
F_A	55036.2700	210.031	
S_B	54826.6190	.38	140.67
F_B	54967.2890	141.05	
S_C	54826.5690	.33	4.451
F_C	54831.0200	4.781	
S_D	54826.5100	.271	209.599
F_D	55036.1090	209.87	
S_E	54826.4600	.221	140.439
F_E	54966.8990	140.66	
S_F	54826.2390	0	4.061
F_F	54830.3000	4.061	
Execution Sequence: $S_F - S_A - S_E - S_D - S_C - S_B - F_F - F_C - F_E - F_B - F_D - F_A$			

Table B.10. Alsys PC AT Ada Compiler Results (Cont'd)

Second Run of Test Case 28 *using Alsys PC AT Ada Compiler, Version 3.2 with a Slice Option of 50 ms.*			
Parameter	Actual Measured Results	Normalized Results	$F_i - S_i$
S_A	55051.8190	.17	209.701
F_A	55261.5200	209.871	
S_B	55052.0390	.39	140.94
F_B	55192.9790	141.33	
S_C	55051.9790	.33	4.51
F_C	55056.4890	4.84	
S_D	55051.9300	.281	209.759
F_D	55261.6890	210.04	
S_E	55051.8690	.22	140.721
F_E	55192.5900	140.941	
S_F	55051.6490	0	4.401
F_F	55056.0500	4.401	
Execution Sequence: $S_F - S_A - S_E - S_D - S_C - S_B - F_F - F_C - F_E - F_B - F_A - F_D$			

Table B.11. Alsys PC AT Ada Compiler Results (Cont'd)

Third Run of Test Case 28 *using Alsys PC AT Ada Compiler, Version 3.2* *with a Slice Option of 50 ms.*			
Parameter	Actual Measured Results	Normalized Results	$F_i - S_i$
S_A	55271.7890	.159	209.761
F_A	55481.5500	209.92	
S_B	55272.0100	.38	140.779
F_B	55412.7890	141.159	
S_C	55271.9600	.33	4.5
F_C	55276.4600	4.83	
S_D	55271.8990	.269	209.76
F_D	55481.6590	210.029	
S_E	55271.8500	.22	140.719
F_E	55412.5690	140.939	
S_F	55271.6300	0	4.39
F_F	55276.0200	4.39	
Execution Sequence: $S_F - S_A - S_E - S_D - S_C - S_B - F_F - F_C - F_E - F_B - F_A - F_D$			

Table B.12. Alsys PC AT Ada Compiler Results (Cont'd)

Actual Results of Running Test Cases 1-9 *using VAX Ada Compiler, Version 1.0* *without the PRAGMA TIME_SLICE*					
Test Case		Actual Measured Results	Normalized Results	$F_i - S_i$	Execution Sequence
1	S_A	44935.5800	0	.39	$S_A - F_A - S_B - F_B$
	F_A	44935.9700	.39		
	S_B	44935.9700	.39	.38	
	F_B	44936.3500	.77		
2	S_A	44941.1900	.02	.38	$S_B - S_A - F_A - F_B$
	F_A	44941.5700	.4		
	S_B	44941.1700	0	.76	
	F_B	44941.9300	.76		
3	S_A	44946.0800	0	.77	$S_A - S_B - F_B - F_A$
	F_A	44946.8500	.77		
	S_B	44946.1000	.02	.38	
	F_B	44946.4800	.4		
4	S_A	44952.2900	0	.4	$S_A - F_A - S_B - F_B$
	F_A	44952.6900	.4		
	S_B	44952.6900	.4	.39	
	F_B	44953.0800	.79		
5	S_A	44958.2200	0	.38	$S_A - F_A - S_B - F_B$
	F_A	44958.6000	.38		
	S_B	44958.6000	.38	.4	
	F_B	44959.0000	.78		
6	S_A	44964.7600	0	.77	$S_A - S_B - F_B - F_A$
	F_A	44965.5300	.77		
	S_B	44964.7800	.02	.37	
	F_B	44965.1500	.39		
7	S_A	44970.0300	.37	.38	$S_B - F_B - S_A - F_A$
	F_A	44970.4100	.75		
	S_B	44969.6600	0	.37	
	F_B	44970.0300	.37		
8	S_A	44974.3100	.02	.37	$S_B - S_A - F_A - F_B$
	F_A	44974.6800	.39		
	S_B	44974.2900	0	.75	
	F_B	44975.0400	.75		
9	S_A	44980.1800	.38	.38	$S_B - F_B - S_A - F_A$
	F_A	44980.5600	.76		
	S_B	44979.8000	0	.38	
	F_B	44980.1800	.38		

Table B.13. VAX Ada Compiler Results

Actual Results of Running Test Cases 10-18 *using VAX Ada Compiler, Version 1.0* *without the PRAGMA TIME_SLICE*					
Test Case		Actual Measured Results	Normalized Results	$F_i - S_i$	Execution Sequence
10	S_A	44996.9300	0	.76	$S_A - F_A - S_B - F_B$
	F_A	44997.6900	.76		
	S_B	44997.6900	.76	.37	
	F_B	44998.0600	1.13		
11	S_A	45007.6600	.02	.75	$S_B - S_A - F_A - F_B$
	F_A	45008.4100	.77		
	S_B	45007.6400	0	1.15	
	F_B	45008.7900	1.15		
12	S_A	45013.1600	0	1.14	$S_A - S_B - F_B - F_A$
	F_A	45014.3000	1.14		
	S_B	45013.1800	.02	.37	
	F_B	45013.5500	.39		
13	S_A	45020.2500	0	.83	$S_A - F_A - S_B - F_B$
	F_A	45021 0800	.83		
	S_B	45021.0800	.83	.39	
	F_B	45021.4700	1.22		
14	S_A	45027.9900	0	.75	$S_A - F_A - S_B - F_B$
	F_A	45028.7400	.75		
	S_B	45028.7400	.75	.39	
	F_B	45029.1300	1.14		
15	S_A	45034.4700	0	1.25	$S_A - S_B - F_B - F_A$
	F_A	45035.7200	1.25		
	S_B	45034.4900	.02	.45	
	F_B	45034.9400	.47		
16	S_A	45055.9700	.39	.76	$S_B - F_B - S_A - F_A$
	F_A	45056.7300	1.15		
	S_B	45055.5800	0	.38	
	F_B	45055.9600	.38		
17	S_A	45062.8900	.02	.77	$S_B - S_A - F_A - F_B$
	F_A	45063.6600	.79		
	S_B	45062.8700	0	1.16	
	F_B	45064.0300	1.16		
18	S_A	45069.3000	.38	.76	$S_B - F_B - S_A - F_A$
	F_A	45070.0600	1.14		
	S_B	45068.9200	0	.38	
	F_B	45069.3000	.38		

Table B.14. VAX Ada Compiler Results (Cont'd)

Actual Results of Running Test Cases 19-27 *using VAX Ada Compiler, Version 1.0* *without the PRAGMA TIME_SLICE*					
Test Case		Actual Measured Results	Normalized Results	$F_i - S_i$	Execution Sequence
19	S_A	45073.7700	0	.38	$S_A - F_A - S_B - F_B$
	F_A	45074.1500	.38		
	S_B	45074.1500	.38	.75	
	F_B	45074.9000	1.13		
20	S_A	45078.1800	.02	.37	$S_B - S_A - F_A - F_B$
	F_A	45078.5500	.39		
	S_B	45078.1600	0	1.21	
	F_B	45079.3700	1.21		
21	S_A	45082.7400	0	1.18	$S_A - S_B - F_B - F_A$
	F_A	45083.9200	1.18		
	S_B	45082.7600	.02	.76	
	F_B	45083.5200	.78		
22	S_A	45087.6700	0	.37	$S_A - F_A - S_B - F_B$
	F_A	45088.0400	.37		
	S_B	45088.0400	.37	.77	
	F_B	45088.8100	1.14		
23	S_A	45092.3800	0	.38	$S_A - F_A - S_B - F_B$
	F_A	45092.7600	.38		
	S_B	45092.7600	.38	.75	
	F_B	45093.5100	1.13		
24	S_A	45097.5700	0	1.14	$S_A - S_B - F_B - F_A$
	F_A	45098.7100	1.14		
	S_B	45097.5900	.02	.75	
	F_B	45098.3400	.77		
25	S_A	45106.6000	.77	.37	$S_B - F_B - S_A - F_A$
	F_A	45106.9700	1.14		
	S_B	45105.8300	0	.76	
	F_B	45106.5900	.76		
26	S_A	45112.9700	.02	.37	$S_B - S_A - F_A - F_B$
	F_A	45113.3400	.39		
	S_B	45112.9500	0	1.16	
	F_B	45114.1100	1.16		
27	S_A	45122.0500	.75	.4	$S_B - F_B - S_A - F_A$
	F_A	45122.4500	1.15		
	S_B	45121.3000	0	.75	
	F_B	45122.0500	.75		

Table B.15. VAX Ada Compiler Results (Cont'd)

First Run of Test Case 28 *using VAX Ada Compiler, Version 1.0* *without the PRAGMA TIME_SLICE*			
Parameter	Actual Measured Results	Normalized Results	$F_i - S_i$
S_A	56519.5700	0	11.97
F_A	56531.5400	11.97	
S_B	56521.7900	2.22	12.59
F_B	56534.3800	14.81	
S_C	56521.7800	2.21	11.77
F_C	56533.5500	13.98	
S_D	56520.8000	1.23	12.66
F_D	56533.4600	13.89	
S_E	56520.3400	.77	11.73
F_E	56532.0700	12.5	
S_F	56520.3300	.76	11.22
F_F	56531.5500	11.98	
Execution Sequence: $S_A - S_F - S_E - S_D - S_C - S_B - F_A - F_F - F_E - F_D - F_C - F_B$			

Table B.16. VAX Ada Compiler Results (Cont'd)

Second Run of Test Case 28 *using VAX Ada Compiler, Version 1.0* *without the PRAGMA TIME_SLICE*			
Parameter	Actual Measured Results	Normalized Results	$F_i - S_i$
S_A	56569.0100	0	11.32
F_A	56580.3300	11.32	
S_B	56571.1100	2.1	11.46
F_B	56582.5700	13.56	
S_C	56571.1000	2.09	11.03
F_C	56582.1300	13.12	
S_D	56570.3200	1.31	11.8
F_D	56582.1200	13.11	
S_E	56569.9100	.9	11.05
F_E	56580.9600	11.95	
S_F	56569.9100	.9	10.43
F_F	56580.3400	11.33	
Execution Sequence: $S_A - S_F - S_E - S_D - S_C - S_B - F_A - F_F - F_E - F_D - F_C - F_B$			

Table B.17. VAX Ada Compiler Results (Cont'd)

Third Run of Test Case 28 *using VAX Ada Compiler, Version 1.0* *without the PRAGMA TIME_SLICE*			
Parameter	Actual Measured Results	Normalized Results	$F_i - S_i$
S_A	56661.1700	0	14.56
F_A	56675.7300	14.56	
S_B	56663.8800	2.71	14.34
F_B	56678.2200	17.05	
S_C	56663.8800	2.71	13.72
F_C	56677.6000	16.43	
S_D	56663.0900	1.92	14.5
F_D	56677.5900	16.42	
S_E	56662.2000	1.03	14.25
F_E	56676.4500	15.28	
S_F	56662.1900	1.02	13.55
F_F	56675.7400	14.57	
Execution Sequence: $S_A - S_F - S_E - S_D - S_C - S_B - F_A - F_F - F_E - F_D - F_C - F_B$			

Table B.18. VAX Ada Compiler Results (Cont'd)

Actual Results of Running Test Cases 1-9 *using VAX Ada Compiler, Version 1.0 with the PRAGMA TIME_SLICE (0.05)*					
Test Case		Actual Measured Results	Normalized Results	$F_i - S_i$	Execution Sequence
1	S_A	47806.4300	0	.73	$S_A - S_B - F_A - F_B$
	F_A	47807.1600	.73		
	S_B	47806.5800	.15	.69	
	F_B	47807.2700	.84		
2	S_A	47818.3400	.02	.5	$S_B - S_A - F_A - F_B$
	F_A	47818.8400	.52		
	S_B	47818.3200	0	1.09	
	F_B	47819.4100	1.09		
3	S_A	47823.1100	0	.98	$S_A - S_B - F_B - F_A$
	F_A	47824.0900	.98		
	S_B	47823.1300	.02	.48	
	F_B	47823.6100	.5		
4	S_A	47831.5500	0	1.19	$S_A - S_B - F_B - F_A$
	F_A	47832.7400	1.19		
	S_B	47831.6200	.07	1.07	
	F_B	47832.6900	1.14		
5	S_A	47836.0900	0	.4	$S_A - F_A - S_B - F_B$
	F_A	47836.4900	.4		
	S_B	47836.4900	.4	.56	
	F_B	47837.0500	.96		
6	S_A	47840.8700	0	.77	$S_A - S_B - F_B - F_A$
	F_A	47841.6400	.77		
	S_B	47840.8900	.02	.38	
	F_B	47841.2700	.4		
7	S_A	47845.9800	.08	.68	$S_B - S_A - F_B - F_A$
	F_A	47846.6600	.76		
	S_B	47845.9000	0	.7	
	F_B	47846.6000	.7		
8	S_A	47850.2500	.02	.69	$S_B - S_A - F_A - F_B$
	F_A	47850.9400	.71		
	S_B	47850.2300	0	1.24	
	F_B	47851.4700	1.24		
9	S_A	47855.8400	.4	.38	$S_B - F_B - S_A - F_A$
	F_A	47856.2200	.78		
	S_B	47855.4400	0	.4	
	F_B	47855.8400	.4		

Table B.19. VAX Ada Compiler Results (Cont'd)

Actual Results of Running Test Cases 10-18 *using VAX Ada Compiler, Version 1.0* *with the PRAGMA TIME_SLICE (0.05)*					
Test Case		Actual Measured Results	Normalized Results	$F_i - S_i$	Execution Sequence
10	S_A	47861.2200	0	1.16	$S_A - S_B - F_B - F_A$
	F_A	47862.3800	1.16		
	S_B	47861.3700	.15	.7	
	F_B	47862.0700	.85		
11	S_A	47865.4000	.02	.77	$S_B - S_A - F_A - F_B$
	F_A	47866.1700	.79		
	S_B	47865.3800	0	1.17	
	F_B	47866.5500	1.17		
12	S_A	47869.6900	0	1.16	$S_A - S_B - F_B - F_A$
	F_A	47870.8500	1.16		
	S_B	47869.7100	.02	.39	
	F_B	47870.1000	.41		
13	S_A	47874.7600	0	1.24	$S_A - S_B - F_B - F_A$
	F_A	47876.0000	1.24		
	S_B	47874.8300	.07	.71	
	F_B	47875.5400	.78		
14	S_A	47879.1400	0	.76	$S_A - F_A - S_B - F_B$
	F_A	47879.9000	.76		
	S_B	47879.9000	.76	.38	
	F_B	47880.2800	1.14		
15	S_A	47882.8400	0	1.15	$S_A - S_B - F_B - F_A$
	F_A	47883.9900	1.15		
	S_B	47882.8600	.02	.37	
	F_B	47883.2300	.39		
16	S_A	47889.5900	.05	1.3	$S_B - S_A - F_B - F_A$
	F_A	47890.8900	1.35		
	S_B	47889.5400	0	.82	
	F_B	47890.3600	.82		
17	S_A	47894.6200	.02	.81	$S_B - S_A - F_A - F_B$
	F_A	47895.4300	.83		
	S_B	47894.6000	0	1.24	
	F_B	47895.8400	1.24		
18	S_A	47898.7800	.42	.85	$S_B - F_B - S_A - F_A$
	F_A	47899.6300	1.27		
	S_B	47898.3600	0	.42	
	F_B	47898.7800	.42		

Table B.20. VAX Ada Compiler Results (Cont'd)

Actual Results of Running Test Cases 19-27 *using VAX Ada Compiler, Version 1.0* *with the PRAGMA TIME_SLICE (0.05)*					
Test Case		Actual Measured Results	Normalized Results	$F_i - S_i$	Execution Sequence
19	S_A	47903.6400	0	.74	$S_A - S_B - F_A - F_B$
	F_A	47904.3800	.74		
	S_B	47903.8000	.16	1.13	
	F_B	47904.9300	1.29		
20	S_A	47908.6600	.02	.39	$S_B - S_A - F_A - F_B$
	F_A	47909.0500	.41		
	S_B	47908.6400	0	1.22	
	F_B	47909.8600	1.22		
21	S_A	47912.8300	0	1.25	$S_A - S_B - F_B - F_A$
	F_A	47914.0800	1.25		
	S_B	47912.8500	.02	.85	
	F_B	47913.7000	.87		
22	S_A	47919.1900	0	.8	$S_A - S_B - F_A - F_B$
	F_A	47919.9900	.8		
	S_B	47919.2600	.07	1.17	
	F_B	47920.4300	1.24		
23	S_A	47924.9900	0	.41	$S_A - F_A - S_B - F_B$
	F_A	47925.4000	.41		
	S_B	47925.4000	.41	.86	
	F_B	47926.2600	1.27		
24	S_A	47931.6800	0	1.21	$S_A - S_B - F_B - F_A$
	F_A	47932.8900	1.21		
	S_B	47931.7000	.02	.82	
	F_B	47932.5200	.84		
25	S_A	47937.3400	.05	.8	$S_B - S_A - F_A - F_B$
	F_A	47938.1400	.85		
	S_B	47937.2900	0	1.2	
	F_B	47938.4900	1.2		
26	S_A	47943.0000	.03	.38	$S_B - S_A - F_A - F_B$
	F_A	47943.3800	.41		
	S_B	47942.9700	0	1.18	
	F_B	47944.1500	1.18		
27	S_A	47949.8300	.82	.4	$S_B - F_B - S_A - F_A$
	F_A	47950.2300	1.22		
	S_B	47949.0100	0	.82	
	F_B	47949.8300	.82		

Table B.21. VAX Ada Compiler Results (Cont'd)

First Run of Test Case 28 *using VAX Ada Compiler, Version 1.0* *with PRAGMA TIME_SLICE (0.05)*			
Parameter	Actual Measured Results	Normalized Results	$F_i - S_i$
S_A	56000.9800	0	12.96
F_A	56013.9400	12.96	
S_B	56001.3500	.37	9.06
F_B	56010.4100	9.43	
S_C	56001.3500	.37	1.7
F_C	56003.0500	2.07	
S_D	56001.2100	.23	12.77
F_D	56013.9800	13	
S_E	56001.1300	.15	9.18
F_E	56010.3100	9.33	
S_F	56001.1300	.15	1.8
F_F	56002.9300	1.95	
Execution Sequence: $S_A - S_F - S_E - S_D - S_C - S_B - F_F - F_C - F_E - F_B - F_A - F_D$			

Table B.22. VAX Ada Compiler Results (Cont'd)

Second Run of Test Case 28 *using VAX Ada Compiler, Version 1.0* *with PRAGMA TIME_SLICE (0.05)*			
Parameter	Actual Measured Results	Normalized Results	$F_i - S_i$
S_A	56022.5000	0	14.04
F_A	56036.5400	14.04	
S_B	56022.8000	.3	9.39
F_B	56032.1900	9.69	
S_C	56022.8000	.3	1.62
F_C	56024.4200	1.92	
S_D	56022.7200	.22	13.68
F_D	56036.4000	13.9	
S_E	56022.6400	.14	9.4
F_E	56032.0400	9.54	
S_F	56022.6400	.14	1.67
F_F	56024.3100	1.81	
Execution Sequence: $S_A - S_F - S_E - S_D - S_C - S_B - F_F - F_C - F_E - F_B - F_D - F_A$			

Table B.23. VAX Ada Compiler Results (Cont'd)

Third Run of Test Case 28 *using VAX Ada Compiler, Version 1.0* *with PRAGMA TIME_SLICE (0.05)*			
Parameter	Actual Measured Results	Normalized Results	$F_i - S_i$
S_A	56042.6700	0	15.32
F_A	56057.9900	15.32	
S_B	56042.9900	.32	9.46
F_B	56052.4500	9.78	
S_C	56042.9900	.32	1.54
F_C	56044.5300	1.86	
S_D	56042.9100	.24	14.98
F_D	56057.8900	15.22	
S_E	56042.8300	.16	9.36
F_E	56052.1900	9.52	
S_F	56042.8300	.16	1.63
F_F	56044.4600	1.79	
Execution Sequence: $S_A - S_F - S_E - S_D - S_C - S_B - F_F - F_C - F_E - F_B - F_D - F_A$			

Table B.24. VAX Ada Compiler Results (Cont'd)

Actual Results of Running Test Cases 1-9 *using Meridian AdaVantage Compiler, Version 2.1*					
Test Case		Actual Measured Results	Normalized Results	$F_i - S_i$	Execution Sequence
1	S_A	45508.5099	9.44	9.4501	$S_B - F_B - S_A - F_A$
	F_A	45517.9600	18.8901		
	S_B	45499.0699	0	9.44	
	F_B	45508.5099	9.44		
2	S_A	45528.8900	0	9.45	$S_A - F_A - S_B - F_B$
	F_A	45538.3400	9.45		
	S_B	45538.3400	9.45	9.44	
	F_B	45547.7800	18.89		
3	S_A	45561.6299	9.4499	9.44	$S_B - F_B - S_A - F_A$
	F_A	45571.0699	18.8899		
	S_B	45552.1800	0	9.4499	
	F_B	45561.6299	9.4499		
4	S_A	45575.4100	0	9.45	$S_A - F_A - S_B - F_B$
	F_A	45584.8600	9.45		
	S_B	45584.9100	9.5	9.45	
	F_B	45594.3600	19.95		
5	S_A	45598.8100	0	9.4499	$S_A - F_A - S_B - F_B$
	F_A	45608.2599	9.449		
	S_B	45608.3100	9.5	9.499	
	F_B	45617.7599	18.9499		
6	S_A	45640.5000	0	9.45	$S_A - F_A - S_B - F_B$
	F_A	45649.9500	9.45		
	S_B	45650.0000	9.5	9.45	
	F_B	45659.4500	18.95		
7	S_A	45679.3300	9.5	9.45	$S_B - F_B - S_A - F_A$
	F_A	45688.7800	18.95		
	S_B	45669.8300	0	9.45	
	F_B	45679.2800	9.45		
8	S_A	45720.5200	9.5	9.45	$S_B - F_B - S_A - F_A$
	F_A	45729.9700	18.95		
	S_B	45711.0200	0	9.45	
	F_B	45720.4700	9.45		
9	S_A	45743.5400	9.5	9.45	$S_B - F_B - S_A - F_A$
	F_A	45752.9900	18.95		
	S_B	45734.0400	0	9.44	
	F_B	45743.4800	9.44		

Table B.25. Meridian AdaVantage Compiler Results

Actual Results of Running Test Cases 10-18 *using Meridian AdaVantage Compiler, Version 2.1*					
Test Case		Actual Measured Results	Normalized Results	$F_i - S_i$	Execution Sequence
10	S_A	47558.8799	9.4499	18.6801	$S_B - F_B - S_A - F_A$
	F_A	47577.5600	28.13		
	S_B	47549.4300	0	9.4499	
	F_B	47558.8799	9.4499		
11	S_A	47586.6200	0	18.67	$S_A - F_A - S_B - F_B$
	F_A	47605.2900	18.67		
	S_B	47605.2900	18.67	9.45	
	F_B	47614.7400	28.12		
12	S_A	47628.6400	9.4501	18.67	$S_B - F_B - S_A - F_A$
	F_A	47647.3100	28.1201		
	S_B	47691.1899	0	9.4501	
	F_B	47628.6400	9.4501		
13	S_A	47651.7599	0	18.6701	$S_A - F_A - S_B - F_B$
	F_A	47670.4300	18.6701		
	S_B	47670.4900	18.7301	9.4499	
	F_B	47679.9399	28.18		
14	S_A	47684.3300	0	18.6799	$S_A - F_A - S_B - F_B$
	F_A	47703.0099	18.6799		
	S_B	47703.0600	18.73	9.4499	
	F_B	47712.5099	28.1799		
15	S_A	47727.0600	0	18.84	$S_A - F_A - S_B - F_B$
	F_A	47745.9000	18.84		
	S_B	47745.9600	18.9	9.44	
	F_B	47755.4000	28.34		
16	S_A	47769.6299	9.6699	18.6701	$S_B - F_B - S_A - F_A$
	F_A	47788.3000	28.34		
	S_B	47759.9600	0	9.67	
	F_B	47769.5800	9.62		
17	S_A	47801.9300	9.51	18.67	$S_B - F_B - S_A - F_A$
	F_A	47820.6000	28.18		
	S_B	47792.4200	0	9.45	
	F_B	47801.8700	9.45		
18	S_A	47834.1700	9.5	18.67	$S_B - F_B - S_A - F_A$
	F_A	47852.8400	28.17		
	S_B	47824.6700	0	9.44	
	F_B	47834.1100	9.44		

Table B.26. Meridian AdaVantage Compiler Results (Cont'd)

Actual Results of Running Test Cases 19-27 *using Meridian AdaVantage Compiler, Version 2.1*					
Test Case		Actual Measured Results	Normalized Results	$F_i - S_i$	Execution Sequence
19	S_A	50889.5600	18.67	9.4499	$S_B - F_B - S_A - F_A$
	F_A	50899.0099	28.1199		
	S_B	50870.8900	0	18.67	
	F_B	50889.5600	18.67		
20	S_A	50903.4000	0	9.45	$S_A - F_A - S_B - F_B$
	F_A	50912.8500	9.45		
	S_B	50912.8500	9.45	18.68	
	F_B	50931.5300	28.13		
21	S_A	50954.7599	18.6799	9.4501	$S_B - F_B - S_A - F_A$
	F_A	50964.2100	28.13		
	S_B	50936.0800	0	18.6799	
	F_B	50954.7599	18.6799		
22	S_A	50968.3300	0	9.44	$S_A - F_A - S_B - F_B$
	F_A	50977.7700	9.44		
	S_B	50977.8300	9.5	18.67	
	F_B	50996.5000	28.17		
23	S_A	51000.6800	0	9.44	$S_A - F_A - S_B - F_B$
	F_A	51010.1200	9.44		
	S_B	51010.1800	9.5	18.67	
	F_B	51028.8500	28.17		
24	S_A	51050.9300	0	9.4499	$S_A - F_A - S_B - F_B$
	F_A	51060.3799	9.4499		
	S_B	51060.4399	9.5099	18.6701	
	F_B	51079.1100	28.18		
25	S_A	51102.4300	18.73	9.45	$S_B - F_B - S_A - F_A$
	F_A	51111.9000	28.18		
	S_B	51083.7200	0	18.68	
	F_B	51102.4000	18.68		
26	S_A	51134.7500	18.73	9.45	$S_B - F_B - S_A - F_A$
	F_A	51144.2000	28.18		
	S_B	51116.0200	0	18.68	
	F_B	51134.7000	18.68		
27	S_A	51166.9900	18.7301	9.4499	$S_B - F_B - S_A - F_A$
	F_A	51176.4399	28.18		
	S_B	51148.2599	0	18.68	
	F_B	51166.9399	18.68		

Table B.27. Meridian AdaVantage Compiler Results (Cont'd)

First Run of Test Case 28 *using Meridian AdaVantage Compiler, Version 2.1*			
Parameter	Actual Measured Results	Normalized Results	$F_i - S_i$
S_A	59526.8700	37.9	244.75
F_A	59771.6200	282.65	
S_B	59517.4200	28.45	235.5199
F_B	59752.9399	263.9699	
S_C	59517.2599	28.2899	226.2401
F_C	59743.5000	254.53	
S_D	59498.5800	9.61	244.75
F_D	59743.3300	254.36	
S_E	59489.1400	.17	235.52
F_E	59724.6600	235.69	
S_F	59488.9700	0	226.19
F_F	59715.1600	226.19	
Execution Sequence: $S_F - S_E - S_D - S_C - S_B - S_A - F_F - F_E - F_D - F_C - F_B - F_A$			

Table B.28. Meridian AdaVantage Compiler Results (Cont'd)

Actual Results of Running Test Cases 1-9 *using Elxsi/Verdix Ada Compiler, Version 5.4*					
Test Case		Actual Measured Results	Normalized Results	$F_i - S_i$	Execution Sequence
1	S_A	47780.416	0	3.0	$S_A - S_B - F_B - F_A$
	F_A	47783.416	3.0		
	S_B	47780.512	.096	2.404	
	F_B	47782.916	2.5		
2	S_A	47790.916	.1	1.4	$S_B - S_A - F_A - F_B$
	F_A	47792.316	1.5		
	S_B	47790.816	0	2.9	
	F_B	47793.716	2.9		
3	S_A	47800.716	0	2.9	$S_A - S_B - F_B - F_A$
	F_A	47803.616	2.9		
	S_B	47800.816	.1	1.4	
	F_B	47802.216	1.5		
4	S_A	47814.120	0	3.3	$S_A - S_B - F_B - F_A$
	F_A	47817.420	3.3		
	S_B	47814.220	.1	2.596	
	F_B	47816.816	2.696		
5	S_A	47826.020	0	1.4	$S_A - F_A - S_B - F_B$
	F_A	47827.420	1.4		
	S_B	47827.420	1.4	1.4	
	F_B	47828.820	2.8		
6	S_A	47836.920	0	2.9	$S_A - S_B - F_B - F_A$
	F_A	47839.820	2.9		
	S_B	47837.020	.1	1.4	
	F_B	47838.420	1.5		
7	S_A	47850.020	1	1.8	$S_B - S_A - F_B - F_A$
	F_A	47851.820	2.8		
	S_B	47849.020	0	2.404	
	F_B	47851.424	2.404		
8	S_A	47859.724	.1	1.4	$S_B - S_A - F_A - F_B$
	F_A	47861.124	1.5		
	S_B	47859.624	0	2.8	
	F_B	47862.424	2.8		
9	S_A	47872.020	1.396	1.404	$S_B - F_B - S_A - F_A$
	F_A	47873.424	2.8		
	S_B	47870.624	0	1.396	
	F_B	47872.020	1.396		

Table B.29. Elxsi/Verdix Ada Compiler Results

Actual Results of Running Test Cases 10-18 *using Elxsi/Verdix Ada Compiler, Version 5.4*					
Test Case		Actual Measured Results	Normalized Results	$F_i - S_i$	Execution Sequence
10	S_A	47886.624	0	4.4	$S_A - S_B - F_B - F_A$
	F_A	47891.024	4.4		
	S_B	47886.724	.1	2.4	
	F_B	47889.124	2.5		
11	S_A	47897.524	.096	3	$S_B - S_A - F_A - F_B$
	F_A	47900.524	3.096		
	S_B	47897.428	0	4.8	
	F_B	47902.228	4.8		
12	S_A	47910.528	0	4.296	$S_A - S_B - F_B - F_A$
	F_A	47914.824	4.296		
	S_B	47910.624	.096	1.404	
	F_B	47912.028	1.5		
13	S_A	47922.528	0	4.404	$S_A - S_B - F_B - F_A$
	F_A	47926.932	4.404		
	S_B	47922.624	.096	2.404	
	F_B	47925.028	2.5		
14	S_A	47935.628	0	2.804	$S_A - F_A - S_B - F_B$
	F_A	47938.432	2.804		
	S_B	47938.432	2.804	1.396	
	F_B	47939.828	4.2		
15	S_A	47953.132	0	4.9	$S_A - S_B - F_B - F_A$
	F_A	47958.032	4.9		
	S_B	47953.232	.1	1.4	
	F_B	47954.632	1.5		
16	S_A	47968.736	1	3.296	$S_B - S_A - F_B - F_A$
	F_A	47972.032	4.296		
	S_B	47967.736	0	2.396	
	F_B	47970.132	2.396		
17	S_A	47978.736	.104	3.7	$S_B - S_A - F_A - F_B$
	F_A	47982.436	3.804		
	S_B	47978.632	0	5.4	
	F_B	47984.036	5.4		
18	S_A	47993.636	1.6	3.2	$S_B - F_B - S_A - F_A$
	F_A	47996.836	4.8		
	S_B	47992.036	0	1.6	
	F_B	47993.636	1.6		

Table B.30. Elxsi/Verdix Ada Compiler Results (Cont'd)

Actual Results of Running Test Cases 19-27 *using Elxsi/Verdix Ada Compiler, Version 5.4*					
Test Case		Actual Measured Results	Normalized Results	$F_i - S_i$	Execution Sequence
19	S_A	48004.440	0	2.396	$S_A - S_B - F_A - F_B$
	F_A	48006.836	2.396		
	S_B	48005.440	1	3.5	
	F_B	48008.940	4.5		
20	S_A	48017.840	.1	1.4	$S_B - S_A - F_A - F_B$
	F_A	48019.240	1.5		
	S_B	48017.740	0	4.5	
	F_B	48022.240	4.5		
21	S_A	48028.640	0	4.3	$S_A - S_B - F_B - F_A$
	F_A	48032.940	4.3		
	S_B	48028.740	.1	2.9	
	F_B	48031.640	3		
22	S_A	48041.448	0	3.4	$S_A - S_B - F_A - F_B$
	F_A	48044.848	3.4		
	S_B	48041.548	.1	4.2	
	F_B	48045.748	4.3		
23	S_A	48053.344	0	1.404	$S_A - F_A - S_B - F_B$
	F_A	48054.748	1.404		
	S_B	48054.748	1.404	2.896	
	F_B	48057.644	4.3		
24	S_A	48066.644	0	4.404	$S_A - S_B - F_B - F_A$
	F_A	48071.048	4.404		
	S_B	48066.748	.104	2.896	
	F_B	48069.644	3		
25	S_A	48079.748	1	2.6	$S_B - S_A - F_A - F_B$
	F_A	48082.348	3.6		
	S_B	48078.748	0	4.8	
	F_B	48083.548	4.8		
26	S_A	48092.952	.1	1.7	$S_B - S_A - F_A - F_B$
	F_A	48094.652	1.8		
	S_B	48092.852	0	5.2	
	F_B	48098.052	5.2		
27	S_A	48112.752	5.8	2.3	$S_B - F_B - S_A - F_A$
	F_A	48115.052	8.1		
	S_B	48106.952	0	5.8	
	F_B	48112.752	5.8		

Table B.31. Elxsi/Verdix Ada Compiler Results (Cont'd)

First Run of Test Case 28 *using Elxsi/Verdix Ada Compiler, Version 5.4*			
Parameter	Actual Measured Results	Normalized Results	$F_i - S_i$
S_A	61406.192	0	57.408
F_A	61463.600	57.408	
S_B	61407.392	1.2	44.912
F_B	61452.304	46.112	
S_C	61407.392	1.2	11.4
F_C	61418.792	12.6	
S_D	61406.392	.2	59.912
F_D	61466.304	60.112	
S_E	61406.296	.104	45.2
F_E	61451.496	45.304	
S_F	61406.296	.104	12.304
F_F	61418.600	12.408	
Execution Sequence: $S_A - S_F - S_E - S_D - S_C - S_B - F_F - F_C - F_E - F_B - F_A - F_D$			

Table B.32. Elxsi/Verdix Ada Compiler Results (Cont'd)

Second Run of Test Case 28 *using Elxsi/Verdix Ada Compiler, Version 5.4*			
Parameter	Actual Measured Results	Normalized Results	$F_i - S_i$
S_A	61467.3040	0	92.904
F_A	61560.2080	92.904	
S_B	61469.0000	1.696	58.008
F_B	61527.0080	59.704	
S_C	61469.0000	1.696	12.304
F_C	61481.3040	14	
S_D	61468.0000	.696	92.608
F_D	61560.6080	93.304	
S_E	61467.9040	.6	58.496
F_E	61526.4000	59.096	
S_F	61467.9040	.6	10.696
F_F	61478.6000	11.296	
Execution Sequence: $S_A - S_F - S_E - S_D - S_C - S_B - F_F - F_C - F_E - F_B - F_D - F_A$			

Table B.33. Elxsi/Verdix Ada Compiler Results (Cont'd)

Third Run of Test Case 28 *using Elxsi/Verdix Ada Compiler, Version 5.4*			
Parameter	Actual Measured Results	Normalized Results	$F_i - S_i$
S_A	61562.0160	0	63
F_A	61625.0160	63	
S_B	61563.2080	1.192	47.704
F_B	61610.9120	48.896	
S_C	61563.2080	1.192	9.104
F_C	61572.3120	10.296	
S_D	61562.2080	.192	60.112
F_D	61622.3200	60.304	
S_E	61562.1040	.088	50.216
F_E	61612.0160	50.304	
S_F	61562.1040	.088	8.208
F_F	61570.3120	8.296	
Execution Sequence: $S_A - S_F - S_E - S_D - S_C - S_B - F_F - F_C - F_E - F_B - F_D - F_A$			

Table B.34. Elxsi/Verdix Ada Compiler Results (Cont'd)

Actual Results of Running Test Cases 1-9 *using Encore/Verdix Concurrent Ada Compiler, Version 5.5*					
Test Case		Actual Measured Results	Normalized Results	$F_i - S_i$	Execution Sequence
1	S_A	42459.449	7.044	6.708	$S_B - F_B - S_A - F_A$
	F_A	42466.157	13.752		
	S_B	42452.405	0	7.038	
	F_B	42459.443	7.038		
2	S_A	42466.805	0	6.709	$S_A - F_A - S_B - F_B$
	F_A	42473.514	6.709		
	S_B	42473.520	6.715	7.035	
	F_B	42480.555	13.75		
3	S_A	42488.749	7.044	6.708	$S_B - F_B - S_A - F_A$
	F_A	42495.457	13.752		
	S_B	42481.705	0	7.037	
	F_B	42488.742	7.037		
4	S_A	42496.725	0	6.707	$S_A - F_A - S_B - F_B$
	F_A	42503.432	6.707		
	S_B	42503.439	6.714	7.034	
	F_B	42510.473	13.748		
5	S_A	42511.704	0	6.71	$S_A - F_A - S_B - F_B$
	F_A	42518.414	6.71		
	S_B	42518.473	6.769	7.035	
	F_B	42525.508	13.804		
6	S_A	42526.208	0	6.71	$S_A - F_A - S_B - F_B$
	F_A	42532.918	6.71		
	S_B	42532.924	6.716	7.036	
	F_B	42539.960	13.752		
7	S_A	42548.164	7.06	6.709	$S_B - F_B - S_A - F_A$
	F_A	42554.873	13.769		
	S_B	42541.104	0	7.038	
	F_B	42548.142	7.038		
8	S_A	42563.169	7.041	6.706	$S_B - F_B - S_A - F_A$
	F_A	42569.875	13.747		
	S_B	42556.128	0	7.035	
	F_B	42563.163	7.035		
9	S_A	42578.166	7.062	6.711	$S_B - F_B - S_A - F_A$
	F_A	42584.877	13.773		
	S_B	42571.104	0	7.038	
	F_B	42578.142	7.038		

Table B.35. Encore/Verdix Concurrent Ada Compiler Results

Actual Results of Running Test Cases 10-18 *using Encore/Verdix Concurrent Ada Compiler, Version 5.5*					
Test Case		Actual Measured Results	Normalized Results	$F_i - S_i$	Execution Sequence
10	S_A	42593.148	7.043	13.441	$S_B - F_B - S_A - F_A$
	F_A	42606.589	20.484		
	S_B	42586.105	0	7.037	
	F_B	42593.142	7.037		
11	S_A	42607.817	0	13.409	$S_A - F_A - S_B - F_B$
	F_A	42621.226	13.409		
	S_B	42621.232	13.415	7.033	
	F_B	42628.265	20.448		
12	S_A	42635.948	7.043	13.412	$S_B - F_B - S_A - F_A$
	F_A	42649.360	20.455		
	S_B	42628.905	0	7.036	
	F_B	42635.941	7.036		
13	S_A	42650.007	0	13.413	$S_A - F_A - S_B - F_B$
	F_A	42663.420	13.413		
	S_B	42663.426	13.419	7.035	
	F_B	42670.461	20.454		
14	S_A	42671.104	0	13.413	$S_A - F_A - S_B - F_B$
	F_A	42684.517	13.413		
	S_B	42684.524	13.42	7.035	
	F_B	42691.559	20.455		
15	S_A	42692.208	0	13.413	$S_A - F_A - S_B - F_B$
	F_A	42705.621	13.413		
	S_B	42705.627	13.419	7.034	
	F_B	42712.661	20.453		
16	S_A	42720.349	7.045	13.414	$S_B - F_B - S_A - F_A$
	F_A	42733.763	20.459		
	S_B	42713.304	0	7.038	
	F_B	42720.342	7.038		
17	S_A	42742.063	7.045	13.417	$S_B - F_B - S_A - F_A$
	F_A	42755.480	20.462		
	S_B	42735.018	0	7.039	
	F_B	42742.057	7.039		
18	S_A	42763.769	7.065	13.411	$S_B - F_B - S_A - F_A$
	F_A	42777.180	20.476		
	S_B	42756.704	0	7.037	
	F_B	42763.741	7.037		

Table B.36. Encore/Verdix Concurrent Ada Compiler Results (Cont'd)

Actual Results of Running Test Cases 19-27 *using Encore/Verdix Concurrent Ada Compiler, Version 5.5*					
Test Case		Actual Measured Results	Normalized Results	$F_i - S_i$	Execution Sequence
19	S_A	42792.487	14.071	6.705	$S_B - F_B - S_A - F_A$
	F_A	42799.192	20.776		
	S_B	42778.416	0	14.064	
	F_B	42792.480	14.064		
20	S_A	42800.505	0	6.713	$S_A - F_A - S_B - F_B$
	F_A	42807.218	6.713		
	S_B	42807.224	6.719	14.075	
	F_B	42821.299	20.794		
21	S_A	42836.579	14.069	6.705	$S_B - F_B - S_A - F_A$
	F_A	42843.284	20.774		
	S_B	42822.510	0	14.062	
	F_B	42836.572	14.062		
22	S_A	42844.624	0	6.708	$S_A - F_A - S_B - F_B$
	F_A	42851.332	6.708		
	S_B	42851.339	6.715	14.064	
	F_B	42865.403	20.779		
23	S_A	42866.704	0	6.718	$S_A - F_A - S_B - F_B$
	F_A	42873.415	6.718		
	S_B	42873.441	6.737	14.066	
	F_B	42887.507	20.803		
24	S_A	42888.833	0	6.708	$S_A - F_A - S_B - F_B$
	F_A	42895.541	6.708		
	S_B	42895.547	6.714	14.062	
	F_B	42909.609	20.776		
25	S_A	42925.015	14.111	6.709	$S_B - F_B - S_A - F_A$
	F_A	42931.724	20.82		
	S_B	42910.904	0	14.069	
	F_B	42924.973	14.069		
26	S_A	42947.097	14.071	6.707	$S_B - F_B - S_A - F_A$
	F_A	42953.804	20.778		
	S_B	42933.026	0	14.065	
	F_B	42947.091	14.065		
27	S_A	42969.191	14.087	6.707	$S_B - F_B - S_A - F_A$
	F_A	42975.898	20.794		
	S_B	42955.104	0	14.068	
	F_B	42969.172	14.068		

Table B.37. Encore/Verdix Concurrent Ada Compiler Results (Cont'd)

First Run of Test Case 28 *using Encore/Verdix Ada Compiler, Version 5.5*			
Parameter	Actual Measured Results	Normalized Results	$F_i - S_i$
S_A	63076.5990	27.895	181.68
F_A	63258.2790	209.575	
S_B	63069.7390	21.035	174.469
F_B	63244.2080	195.504	
S_C	63069.5920	20.888	167.738
F_C	63237.3300	188.626	
S_D	63055.8670	7.163	181.298
F_D	63237.1650	188.461	
S_E	63048.8460	.142	174.706
F_E	63223.4100	174.706	
S_F	63048.7040	0	167.665
F_F	63216.3690	167.665	
Execution Sequence: $S_F - S_E - S_D - S_C - S_B - S_A - F_F - F_E - F_D - F_C - F_B - F_A$			

Table B.38. Encore/Verdix Ada Compiler Results (Cont'd)

Second Run of Test Case 28 *using Encore/Verdix Ada Compiler, Version 5.5*			
Parameter	Actual Measured Results	Normalized Results	$F_i - S_i$
S_A	63286.8750	27.871	181.722
F_A	63468.5970	209.593	
S_B	63280.0220	21.018	174.497
F_B	63454.5190	195.515	
S_C	63279.8760	20.872	167.763
F_C	63447.6390	188.635	
S_D	63266.1680	7.164	181.305
F_D	63447.4730	188.469	
S_E	63259.1460	.142	174.584
F_E	63433.7300	174.726	
S_F	63259.0040	0	167.689
F_F	63426.6930	167.689	
Execution Sequence: $S_F - S_E - S_D - S_C - S_B - S_A - F_F - F_E - F_D - F_C - F_B - F_A$			

Table B.39. Encore/Verdix Ada Compiler Results (Cont'd)

Third Run of Test Case 28 *using Encore/Verdix Ada Compiler, Version 5.5*			
Parameter	Actual Measured Results	Normalized Results	$F_i - S_i$
S_A	63497.1770	27.873	181.68
F_A	63678.8570	209.553	
S_B	63490.3260	21.022	174.452
F_B	63664.7780	195.474	
S_C	63490.1790	20.875	167.72
F_C	63657.8990	188.595	
S_D	63476.4660	7.162	181.267
F_D	63657.7330	188.429	
S_E	63469.4460	.142	174.541
F_E	63643.9870	174.683	
S_F	63469.3040	0	167.644
F_F	63636.9480	167.644	
Execution Sequence: $S_F - S_E - S_D - S_C - S_B - S_A - F_F - F_E - F_D - F_C - F_B - F_A$			

Table B.40. Encore/Verdix Ada Compiler Results (Cont'd)

Bibliography

1. A., Burns and A.J. Wellings. "Real-Time Ada Issues," *ACM SIGADA Ada Letters, 1987 Special Edition*, *VII*(6):43–46 (Fall 1987).

2. Alsys Inc., Waltham, MA. *Alsys PC AT Ada Compiler User's Guide* (Version 3.2 Edition). August 1987.

3. Booch, Grady. *Software Engineering with Ada* (Second Edition). Menlo Park, CA.: The Benjamin/Cummings Publishing Company, 1987.

4. Borger, Mark and others. "A Testbed for Investigating Real-Time Ada Issues," *ACM SIGADA Ada Letters, 1988 Special Edition*, *VIII*(7):7–11 (Fall 1988).

5. Burger, Thomas M. and Kjel' W. Nielson. "An Assessment of the Overhead Associated with Tasking Facilities and Task Paradigms in Ada," *ACM SIGADA Ada Letters*, *VII*(1):49–58 (Jan/Feb 1987).

6. Coffman, Edward G. and Leonard Kleinrock. "Computer Scheduling Methods and Their Countermeasures." In *Spring Joint Computer Conference*, pages 11–21, Vol 32 1968.

7. Cornhill, Dennis. "Session Summary: Tasking," *ACM SIGADA Ada Letters, 1987 Special Edition*, *VII*(6):29–32 (Fall 1987).

8. Cornhill, Dennis and others. "Limitations of Ada for Real-Time Scheduling," *ACM SIGADA Ada Letters, 1987 Special Edition*, *VII*(6):33–39 (Fall 1987).

9. Cornhill, Dennis and Lui Sha. "Priority Inversion in Ada," *ACM SIGADA Ada Letters*, *VII*(7):30–32 (Nov/Dec 1987).

10. Deitel, Harvey M. *An Intoduction to Operating Systems*. Reading, Mass.: Addison-Wesley Publishing Company, Inc., 1984.

11. Digital Equipment Corporation, Maynard, MA. *VAX Ada Language Reference Manual* (Version 1.0 Edition), February 1985.

12. DoD. *Military Standard: Ada Programming Language - ANSI/MIL-STD-1815A*. Department of Defense, Washington, D.C., January 1983.

13. Finkel, Raphael A. *An Operating Systems Vade Mecum*. Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1986.

14. Frankel, Gary. "Improving Ada Tasking Performance," *ACM SIGADA Ada Letters, 1987 Special Edition*, *VII*(6):47–48 (Fall 1987).

15. Gonzales, Jr., M. J. "Deterministic Processor Scheduling," *ACM Computing Surveys*, *9*(3):177–182 (Sep 1977).

16. Goodenough, John B. and Lui Sha. "The Priority Ceiling Protocol: A Method for Minimizing the Blocking of High Priority Ada Tasks," *ACM SIGADA Ada Letters, 1988 Special Edition*, *VIII*(7):20–31 (Fall 1988).

17. Habermann, A. N. *Introduction to Operating System Design*. Chicago, Ill.: Science Research Associates, Inc., 1976.

18. Kleinrock, Leonard. "A Continuum of Time-Sharing Scheduling Algorithms." In *Spring Joint Computer Conference*, pages 453–458, Vol 36 1970.

19. Levine, Gertrude. "The Control of Priority Inversion in Ada," *ACM SIGADA Ada Letters*, *VIII*(6):53–56 (Nov/Dec 1988).

20. Liu, Jane W.S. and Kwei-Jay Lin. "On Means to Provide Flexibility in Scheduling," *ACM SIGADA Ada Letters, 1988 Special Edition*, *VIII*(7):32–34 (Fall 1988).

21. Locke, C. Douglass. and others. "Priority Inversion and Its Control: An Experimental Investigation," *ACM SIGADA Ada Letters, 1988 Special Edition*, *VIII*(7):39–42 (Fall 1988).

22. Locke, C. Douglass. and David R. Vogel. "Problems in Ada Runtime Task Scheduling," *ACM SIGADA Ada Letters, 1987 Special Edition*, *VII*(6):51–53 (Fall 1987).

23. McCormick, Frank. "Scheduling Dificulties of Ada in the Hard Real-Time Environment," *ACM SIGADA Ada Letters, 1987 Special Edition*, *VII*(6):49–50 (Fall 1987).

24. Serlin, Omri. "Scheduling of Time Critical Processes." In *Spring Joint Computer Conference*, pages 925–932, Vol 40 1972.

25. Silberschatz, Abraham and James L. Peterson. *Operating System Concepts*. Reading, Mass.: Addison-Wesley Publishing Company, Inc., 1988.

26. Verdix Corporation, San Hose, CA. *Elxsi/Verdix Ada Development System Reference* (Version 5.4 Edition), 1988.

27. Whitehill, Stephen B. and others. *Meridian AdaVantage Compiler User's Guide* (Version 2.1 Edition). Meridian Software Systems, Inc., Laguna Hills, CA.

Vita

Captain Gary A. Whitted [redacted] He graduated from Franklin High School in Franklin, Wisconsin, in 1972. He enlisted in the United States Air Force on 7 November 1973, attended Basic Military Training School at Lackland AFB, Texas for six weeks, and then transferred to Keesler AFB, Mississippi, where he completed Electronic Equipment Repair training. Next, he was assigned to Patrick AFB, Florida from August 1974 until March 1980 where he worked as a Ground Radio Equipment Repairman in the 2179 Communications and Installation Group. In September 1979, he was selected for the Airman Education Commissioning Program (AECP) and was transferred to the University of Florida in Gainesville. He received the degree of Bachelor of Science in Engineering (Computer and Information Sciences) in July 1982. Then, he attended Officer Training School (OTS) at Lackland AFB and received a commission on 13 October 1982. From OTS, he was assigned to Wright-Patterson AFB, Ohio where he served as an Information Systems Requirements and Plans Officer at the Aeronautical Systems Division's Information Systems and Technology Center from October 1982 until November 1985. He remained at WPAFB and transferred to the C-17 System Program Office (SPO) where he served as a Software Design Manager until entering the School of Engineering, Air Force Institute of Technology, in May 1988. While serving in the C-17 SPO and during his first year at AFIT, he also completed coursework at Wright State University in Fairborn, Ohio and received the degree of Masters in Business Administration in June 1989. Captain Whitted is married and has 3 children.

[redacted]

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY 2b. DECLASSIFICATION / DOWNGRADING SCHEDULE	3. DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; distribution unlimited
4. PERFORMING ORGANIZATION REPORT NUMBER(S) AFIT/GCS/ENG/89D-18	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION School of Engineering 6b. OFFICE SYMBOL *(If applicable)* AFIT/ENA	7a. NAME OF MONITORING ORGANIZATION
6c. ADDRESS *(City, State, and ZIP Code)* Air Force Institute of Technology Wright-Patterson AFB, OH 45433	7b. ADDRESS *(City, State, and ZIP Code)*
8a. NAME OF FUNDING / SPONSORING ORGANIZATION Ada Language Control Facility 8b. OFFICE SYMBOL *(If applicable)* ASD/SCEL	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS *(City, State, and ZIP Code)* ASD Communications-Computer Systems Center Wright-Patterson AFB, OH 45433	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO / WORK UNIT ACCESSION NO.

11. TITLE *(Include Security Classification)* (UNCLASSIFIED)
Determination of the Underlying Task Scheduling Algorithm for an Ada Run-Time System

12. PERSONAL AUTHOR(S)
Gary A Whitted, Capt, USAF

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT *(Year, Month, Day)*	15. PAGE COUNT
MS Thesis	FROM ____ TO ____	1989, December	126

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*
FIELD	GROUP	SUB-GROUP	Software Engineering, Ada task scheduling, Computer Programming, Ada Compilers, Algorithms
12	05		

19. ABSTRACT *(Continue on reverse if necessary and identify by block number)*

Thesis Advisor: James W. Howatt, Maj, USAF
Assistant Professor of Computer Systems

Abstract:
The purpose of this thesis investigation was to determine whether the task scheduling algorithm of an Ada compiler could be detected using a suite of Ada programs. This was done by identifying the task parameters and algorithm characteristics which differentiate one scheduling algorithm from the others. After these parameters and characteristics were identified , a set of test cases was developedto encompass the various parameter relationships required to detect the execution of individual algorithms. These test cases were modeled using Ada programs. Then, the programs were compiled and executed using several Ada compilers where the task scheduling algorithms of five run-time systems was known. The execution results were analyzed to determine whether the Ada programs were capable of revealing the task scheduling algorithm used by the Ada run-time system. The analyses showed that the detection of five scheduling schemes is possible using a single Ada program.

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT ☐ UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL James W. Howatt, Maj, USAF	22b. TELEPHONE *(Include Area Code)* 513-255-3576 22c. OFFICE SYMBOL AFIT/ENG